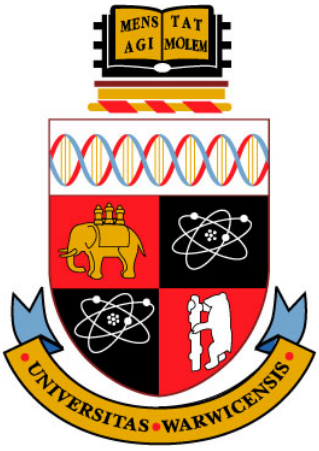


Towards Effective Dynamic Resource Allocation for Enterprise Applications

by

Adam Peter Chester

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

September 2011

Abstract

The growing use of online services requires substantial supporting infrastructure. The efficient deployment of applications relies on the cost effectiveness of commercial hosting providers who deliver an agreed quality of service as governed by a service level agreement for a fee. The priorities of the commercial hosting provider are to maximise revenue, by delivering agreed service levels, and minimise costs, through high resource utilisation.

In order to deliver high service levels and resource utilisation, it may be necessary to reorganise resources during periods of high demand. This reorganisation process may be manual or alternatively controlled by an autonomous process governed by a dynamic resource allocation algorithm. Dynamic resource allocation has been shown to improve service levels and utilisation and hence, profitability.

In this thesis several facets of dynamic resource allocation are examined to asses its suitability for the modern data centre. Firstly, three theoretically derived policies are implemented as a middleware for a modern multi-tier Web application and their performance is examined under a range of workloads in a real world test bed.

The scalability of state-of-the art resource allocation policies are explored in two dimensions, namely the number of applications and the quantity of servers under control of the resources allocation policy. The results demonstrate that

current policies presented in the literature demonstrate poor scalability in one or both of these dimensions. A new policy is proposed which has significantly improved scalability characteristics and the new policy is demonstrated at scale through simulation.

The placement of applications in across a datacenter makes them susceptible to failures in shared infrastructure. To address this issue an application placement mechanism is developed to augment any dynamic resource allocation policy. The results of this placement mechanism demonstrate a significant improvement in the worst case when compared to a random allocation mechanism.

A model for the reallocation of resources in a dynamic resource allocation system is also devised. The model demonstrates that the assumption of a constant resource reallocation cost is invalid under both physical reallocation and migration of virtualised resources.

Dedication

This thesis is dedicated to Louise for her love and support.

Acknowledgements

I am grateful to many people for their assistance during the four years it has taken me to develop my PhD. Firstly my supervisor, Professor Stephen Jarvis for his advice and direction which has guided me towards completion of this thesis. I would also like the thank Dr. Arshad Jhumka, for his enthusiasm for new ideas and working with me to develop some of the work in this thesis.

Special thanks also go Matthew Leeke and Simon Hammond, for their encouragement, advice and general banter in the lab. Other members of the Performance Computing and Visualisation group past and present including Mohammed Al-Ghamdi, James Davies, Henry Franks, Dr Nathan Griffiths, Dr Ligang He, Dr. Gihan Mudilage, Simon Penneycook, Oliver Perks, Jessica Smith, Steven Wright and last but not least Dr. James Xue, who have helped in countless ways over the four years.

Outside of DCS I'd like to thank Charlotte, James, Claire and Phil Gallagher for the fun distractions on Tuesday mornings and giving me somewhere to escape to clear my head. I'd also like to thank the Leam family in its entirety for the many dinners and pub visits.

My family Mum, Dad, Tara, Hope and Dean have been amazingly supportive and encouraging during my PhD studies.

Finally, I'd like to thank Louise for being the best friend I could wish for and making it possible for me to study. Thanks for waiting, I know you've put up with me working a lot to get to this point.

Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work described in this thesis has been undertaken by myself except where otherwise stated. Portions of this work have been published in the following publications:

- **A.P. Chester**, J.W.J. Xue, L. He and S.A. Jarvis. A System For Dynamic Server Allocation in Application Server Clusters. UK Performance Engineering Workshop, London, UK. 2008.
- **A.P. Chester**, J.W.J. Xue, L. He and S.A. Jarvis. A System For Dynamic Server Allocation in Application Server Clusters. In 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications, Sydney, Austrailia, 2008.
- J.W.J. Xue, **A.P. Chester**, L. He and S.A. Jarvis. Dynamic Resource Allocation in Enterprise Systems. International Conference on Parallel and Distributed Systems. Melbourne, Australia. 2008.
- J.W.J. Xue, **A.P. Chester**, L. He and S.A. Jarvis. Model-Driven Server

Allocation in Distributed Enterprise Systems. International Conference on Adaptive Business Information Systems. Leipzig, Germany. 2009.

- J.W.J. Xue, **A.P. Chester**, L. He and S.A. Jarvis. Model-Driven Server Allocation in Distributed Enterprise Systems. Communications of SiWN, 6, pp 42-5.0. 2009

- M. Al-Ghamdi, **A.P. Chester** and S.A. Jarvis. Predictive and Dynamic resource Application for Enterprise Applications. In 10th International Conference on Scalable Computing and Communications 2010.

- M. Al-Ghamdi, **A.P. Chester**, L.He and S.A. Jarvis. Dynamic Resource Allocation and Active Predictive Models for Enterprise Applications. In Proceedings of the 1st International Conference on Cloud Computing and Services Science, Noordwijkerhout, The Netherlands. 2010

- **A.P. Chester**, M. Leeke, M. Al-Ghamdi, S.A. Jarvis and A. Jhumka. A Modular Failure-Aware Resource Allocation Architecture for Cloud Computing. UK Performance Engineering Workshop, Bradford, UK. 2011.

- **A.P. Chester**, M. Leeke, M. Al-Ghamdi, A. Jhumka and S.A. Jarvis. A Framework for Data Centre Scale Dynamic Resource Allocation Algorithms. In 11th IEEE International Conference on Scalable Computing and Communications, Pafos, Cyprus. 2011.

- M. Al-Ghamdi, **A.P. Chester**, L.He and S.A. Jarvis. Dynamic Active Window Management: A Method for Improving Revenue Generation in Dynamic Enterprise Systems. In 11th IEEE International Conference on Scalable Computing and Communications, Pafos, Cyprus. 2011.

- D.A. Bacigalupo, J.I. van Hemert, X. Chen, A. Usmani, **A.P. Chester**, L. He, D.N. Dillenberger, G.B. Wills, L. Gilbert, S.A. Jarvis: Managing dynamic enterprise and urgent workloads on clouds using layered queuing and historical performance models. Simulation Modelling Practice and Theory 19(6): 1479-1495 (2011)

Sponsorship and Grants

This work is supported in part by the UK Engineering and Physical Science Research Council (EPSRC) contract number EP/C538277/1, "Dynamic Operating Policies for Commercial Hosting Environments". This project involved collaboration with IBM, HP Labs, the University of Newcastle and the National Business to Business Centre.

There has been additional collaboration with Deutsche Bank in the form of capacity planning for a large high-volume application.

Abbreviations

DRA - Dynamic resource allocation

GB - Gigabyte

GHz - Gigahertz

MB - Megabyte

mbps - Megabits per second

MPI - Message passing interface

QoS - Quality of service

RDBMS - Relational database managment software

SLA - Service level agreement

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The complexity and scale of enterprise systems has grown significantly with the increase in network availability. This has increased the workload to which enterprise-hosted applications are subjected, and has led to an expansion in the infrastructure required to support them.

Modern enterprise systems must deliver high performance and reliability in addition to meeting functional requirements. Capacity planning for online services can be a difficult task due to the potential for sudden, and often unexpected, spikes in network traffic. These spikes may be several orders of magnitude greater than the regular load. In such cases the performance of the service may degrade to the point of being unusable. Outages such as these can have very real consequences for the reputation of the hosted service and its potential revenue. Where the planned capacity far exceeds that which is required, there is a clear cost inefficiency as more resource is provisioned than required.

To mitigate against the capital expenditure involved building a data centre, companies often outsource the hosting of services to a commercial hosting

provider. The hosting provider agrees to deliver a certain quality of service (QoS), as defined by a service level agreement (SLA), in exchange for a given fee. The hosting provider pays a penalty when the service level agreement is violated. The goal of the hosting provider is to maximise profit, by avoiding SLA violations, and minimise costs by ensuring high utilisation.

1.1 Motivation and Problem Statement

Where a hosting provider dedicates resources to a customer the scale of the resources available is fixed, and may be subject to under or over provisioning. This may lead to costly SLA violations. Dynamic resource allocation has been shown to improve service levels and balance resource utilisation across available resources for hosted applications [14], making it attractive to hosting providers.

Despite the potential benefits of dynamic resource allocation it has had limited uptake by modern hosting platforms. There are many issues affecting the implementation of these policies. Much of the current work on dynamic resource allocation focusses exclusively on the performance of applications or the optimising the profitability of the overall platform. Research into these platforms has to date been conducted in small-scale environments without consideration towards resource failures and migration cost.

This thesis aims to address issues affecting the adoption of dynamic resource allocation in commercial hosting environments.

1.2 Thesis Contributions

The overarching contribution of this thesis is a suite of algorithms and models that support the practical deployment of dynamic resource allocation in enterprise systems. We do this through the following contributions:

- We develop a real-world testbed for the evaluation of theoretically derived switching policies. The policies are tested using synthetic workloads and

the behaviour of the policies is evaluated.

- A proof of the exponential complexity of state-of-the-art policies, and an assessment of their scalability characteristics when scaled across both resources and applications. A new heuristic policy is developed which offers improved scalability and an instance of the framework is developed as a policy to demonstrate the improvement against state-of-the-art policies.

- A robustness metric, *CapacityLoss*, is developed to measure the potential impact of infrastructure failures to applications. We consider infrastructure failures at the rack level. A failure-aware allocator is developed as a modular component to provision resources that minimise the potential *CapacityLoss* from a server allocation. The allocator can be used to augment any resource allocation policy as it operates independently of the policy.

- We develop a cost model for the reallocation of resources to inform better decision making in resource allocation policies. The cost model is evaluated in dedicated and virtualised hosting environments and shown to be accurate to within 11% in both cases.

1.3 Thesis Structure

The contents of this chapter detail the motivation and problem which this thesis aims to address. The remainder of this thesis is structured as follows:

Chapter 2 introduces the concepts of enterprise applications and dynamic resource allocation, and also discusses their evolution and the current state of the art. It also introduces a number of key issues with introducing dynamic resource allocation into large-scale environments.

Chapter 3 presents the tools used throughout this thesis for the measurement

and assessment of dynamic resource allocation policies. This chapter describes the software stack that is written for small-scale evaluation and the simulator which has been developed to test ideas at scale. Validations are provided through a real-world performance study with a commercial partner.

Chapter 4 is primarily concerned with the scalability of resource allocation policies. Many of the proposed policies and heuristics consider scalability in a single dimension, which is usually the amount of resource to be allocated. In this chapter we analyse the impact of scaling an additional dimension, namely the number of applications requiring resource.

Chapter 5 addresses the issue of resource failures on dynamic resource allocation. Active and passive failure detection mechanisms are examined, and a failure aware approach is developed to mitigate the risk of rack failures through the balanced allocation of resources.

Chapter 6 assess the cost the system of resource migration for the purposes of providing accurate duration for the reallocation of multiple resources. The chapter explores the impact of large-scale resource reallocation on reallocation duration.

Finally, Chapter 7 provides a conclusion of the research presented in the thesis and presents future directions for the current work.

CHAPTER 2

Review of Enterprise Application Architecture

Enterprise application architectures have seen a number of changes from their initial conception. Early applications were designed as monolithic batch processing systems designed for a specific deployment architecture, consisting of a combined hardware and software platform. This is a stark contrast to today, where an apparently unlimited resource may be provisioned on demand.

In this chapter we provide a description of (1) an enterprise application architecture as used in this thesis, (2) some of the common metrics used in the performance evaluation of these systems, (3) dynamic resource allocation and recent advances in cloud computing and finally (4) issues in dynamic resource provisioning.

2.1 Multi-tier Architecture

The multi-tier architecture (sometimes referred to as the n-tier architecture) provides a pattern for the design of enterprise applications [37]. The arrange-

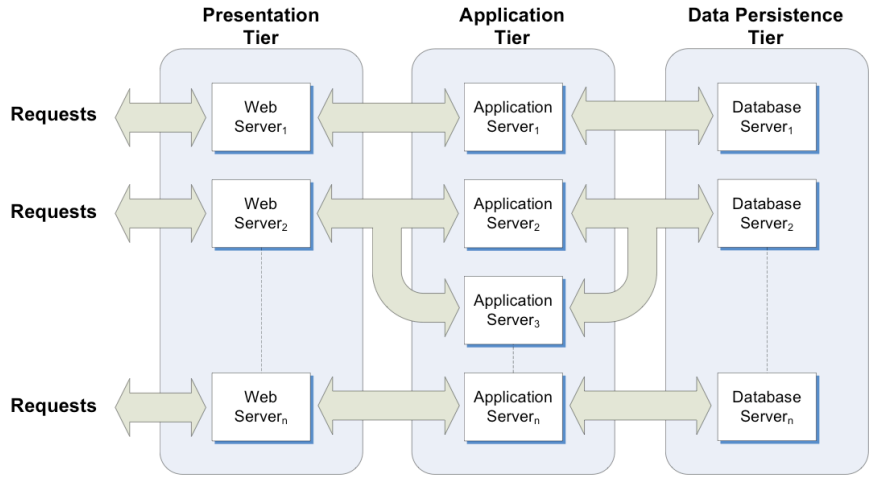


Figure 2.1: 3-tier enterprise application architecture.

ment allows for an applications functionality to be designed in a modular way, which is good practice as it allows for individual modules to be replaced during the application's lifetime rather than requiring full redevelopment.

Some applications may require a single tier for application logic. Other services may be provided by two tiers with a tier for application logic and a tier solely responsible for data storage. Web applications are commonly mapped onto three tiers [35] where each tier has a specific responsibility. The tiers involved are a presentation tier, an application tier and a data persistence tier. In this thesis we consider three tier web applications which are shown in Figure 2.1.

The presentation tier acts as the entry point for an application, forwarding requests to the application tier where required. This tier may also serve static application assets such as images and non-dynamic application content. The application tier implements the business logic for an application. This tier executes the core processing for the application, and may provide services such as connection pooling for database access. The introduction of managed storage, in the form of general purpose Relational Database Management Systems

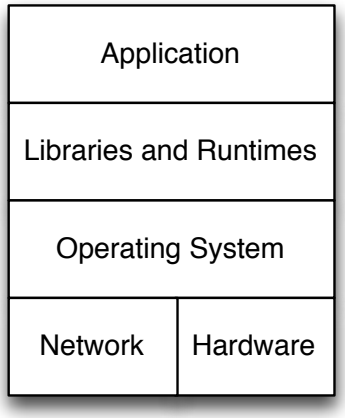


Figure 2.2: Full software and hardware stack.

(RDBMS), has enabled the separation of application logic from its data persistence. Additionally storing data in an RDBMS and accessing it via a standard interface allows multiple applications to access the data simultaneously. These RDBMS have become complex software themselves, requiring expert knowledge to maintain and operate. This tier requires fast I/O subsystems to access data not stored in memory. Each tier may be served by a single server or a cluster of servers to improve performance through horizontal scaling [67].

The nodes at each tier are comprised of multiple hardware and software components, which can be seen in figure 2.2. The application may be supported by other software processes, libraries or runtimes which are in turn executed by an operating system. The operating system interacts directly with the physical hardware and external networking interfaces.

In such an environment the underlying network model is typically an Ethernet network for communication between nodes, however specialised interconnects may be used for certain applications, e.g. distributed file storage in the form of a SAN. The networking within a data centre commonly consists of a fat tree topology [8], with the number of tiers in the tree dependant on the number of hosts to be supported.

A request to an application may require services at a single tier or multiple

tiers, depending on its type. In the case of a request for static assets (images or css files) the Web server may return these files without the need to invoke the services of the underlying tiers. In the case of a dynamic request (a request which requires some result to be computed), the web server forwards the request on to an application server. The application server is responsible for the processing of the request and its subsequent rendering into a suitable format. The application server may issue read or write data to the database server as required to perform the request. Once processing has been completed the application server renders its response to the Web server which then forwards it to the client.

2.2 Performance Evaluation of Enterprise Systems

Understanding the performance characteristics of an application is essential in order to effectively plan the required capacity for the system.

Performance studies of enterprise applications may be conducted in two main ways- via an empirical study of the system or through performance modelling. Measuring the performance of the system in deployment gives accurate results for a given system, executing on a given platform. In the case of performance tuning, this works well as an iterative process as the system can be tested before and after tuning for comparison. The downside to this approach is the lack of portability between applications and deployment platforms [58].

Performance modelling of an application may take two main forms; an analytical model or a simulation model. Analytical modelling of a system develops a mathematical model of a system expressed as a set of equations. Modifying the parameters of the model yields performance results for a given situation. In enterprise systems, analytical models are often based on queueing theory [71, 49] and queuing networks [18]. Where models may be expressed in a simple form, execution and re-evaluation of the model may be near instantaneous. However in large multi-class queuing networks, iterative techniques (such as mean-value

analysis [57]) become cumbersome.

Simulation modelling of enterprise systems [48] can be useful as it can be used to model platforms in which proper system equations are too difficult to derive. Additionally, simulation can capture characteristics of the system which cannot be easily modelled through a queueing model, e.g., network contention, the main downside to the use of simulation is a potentially long execution time and high memory consumption.

A performance model is valid if the results given by the model are within an acceptable range of the system. Work in [52] suggests that performance models with an accuracy of between 10 and 30% are acceptable for capacity planning.

2.2.1 Performance Metrics and Optimisation

There are many possible metrics which can be used to measure the performance of a system. Work in [51] suggest the following as the most common measurements for Web applications:

- End-to-end response time. This is the time taken from a client issuing a request to a system, and the complete response returning. It considers all required processing and network propagation time.
- Site response time. Time taken for a site to respond to a received request. This ignores network communication between the client and the server.
- Request throughput. This is the number of requests served in a unit of time, usually seconds.
- Network throughput. The amount of data transferred per unit time, usually expressed as Mbps.
- Errors per second. This considers the number of requests not served successfully by the service.
- Visitors per day. The number of pages requested per day.

- Unique visitors per day. The number of unique visitors per day who visit one or more pages.

The work in this thesis primarily uses end-to-end response time and the number of failed requests as the main performance metric for applications being tested.

Performance optimisation for single application environments has been extensively studied [21, 27, 32, 34, 50]. [21, 34, 62] focus on request scheduling strategies for performance optimisation.

2.3 Outsourced Hosting

Organisations may outsource their service hosting infrastructure to dedicated hosting providers. Dedicated hosts are able to provide redundancy in power and network capacity and allow organisations to avoid capital expenditure on physical infrastructure.

For enterprise applications hosted externally, service level agreements (SLAs) attempt to capture the desired quality of service (QoS). Typically a SLA will normally include expected response times for an application and the availability required. The cost of the provided hosting will depend on the agreed service level, with higher service levels commanding a higher premium.

The work in this thesis is designed to support infrastructure providers in improving resource utilisation, reducing the possibility of SLA violation and, in so doing, increase profitability.

2.4 Dynamic Resource Allocation

As the usage patterns of many applications have migrated from batch processing to online processing, the demands placed on applications have changed significantly. Usage patterns of online applications have been shown to be periodic at a number of scales [30, 81] however, online applications are susceptible to rapid changes in popularity where peak demand for a service may be orders of

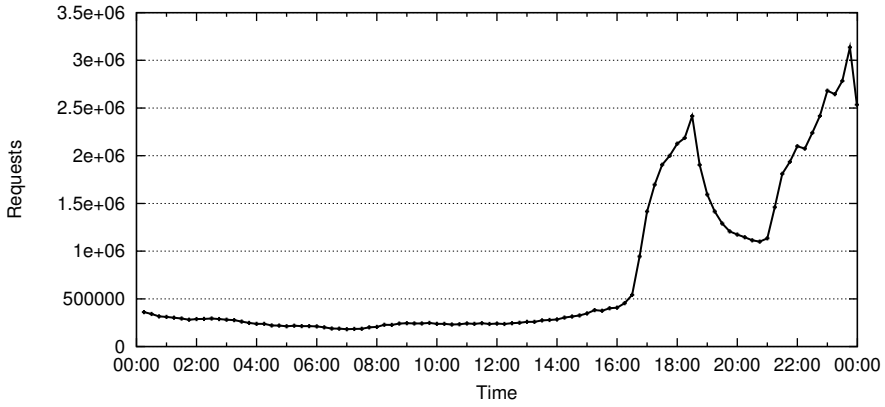


Figure 2.3: Workload variation in 1998 World Cup Web traffic.

magnitude greater than the average. A study of the 1998 World Cup workload [15] demonstrates this rapid change in workload (see Figure 2.3). This workload is commonly referred to in the literature as it is widely available.

The traffic sample covers the full 24 hour period of the 30^{th} June 1998. On this day two matches are played the first at 16:30 and the second at 21:00. The peaks in traffic correspond directly the matches. Note that traffic to the site is low, relative to the peak, prior to 16:00.

An early form of dynamic resource allocation was the use of priority queues to offer differentiated services to different classes of request to optimise company revenue [53]. In this work different priorities are assigned to requests based on their importance.

The main premise of dynamic resource allocation is that resources can be made available to applications experiencing high demand, to maintain a QoS and avoid a site becoming unresponsive. In the presence of variable workload, dynamic provisioning of server resources has been shown to demonstrate improvements to QoS and resource utilisation in an online environment [13, 29, 71]. Many different resource allocation policies have been proposed [65, 80], each with different approaches and results. Work in [25] demonstrates the suitability of dynamic resource allocation for handling flash crowds.

2.5 The Advent of Cloud Computing

In recent years cloud computing has become a viable alternative to capital expenditure on computational infrastructure[16]. Cloud computing platforms have been enabled by developments in virtualisation software and hardware support added to commodity hardware by vendors.

Through virtualisation many applications may be consolidated onto fewer physical servers, reducing capital expenditure to businesses or increasing client density in the case of hosting providers. Modern hypervisors allow a fine grained approach to resource management, allowing a guaranteed minimum level of service but also the ability to burst above a quota where spare capacity exists.

Though in its infancy, cloud computing has differentiated itself into three main categories [73], each offering a differing degree of flexibility.

- Infrastructure as a Service (IaaS) provides physical support for virtualised servers in the form of combined storage, networking and an execution platform. A provider may offer a bare operating system as a starting point or allow a customer to provide their own virtual machine images.

- Platform as a Service (PaaS) offers a platform for the hosting of an application or service. This level of cloud abstraction allows application deployment without the need to manage and administrate the underlying hardware.

- Software as a Service (SaaS) delivers software to the user via the internet, without requiring installation. All data is stored on servers hosted by the application. This approach makes the application available to a user around the world.

This new operational paradigm has increased the potential for companies to outsource their infrastructure and, in so doing, led to a number of commercial clouds [1, 7]. While these cloud platforms have been shown to be relatively stable, they have suffered from a number of high profile outages [59]. These

outages can affect a wide range of applications which rely on the leased services. To mitigate against the risks of a single provider organisations may choose to use multiple providers together in a federated cloud. There are a number of technical challenges to enable this however, work in [24] suggests a mechanism for handling this process.

The IaaS category of cloud computing is most similar to the environment we present in Section 2.1 as it allows virtualised hardware and network resources to be leased and used for any purpose. The issue for the cloud provider is to manage the clients' requirements across the available physical capacity of the system.

2.6 Issues around Dynamic Resource Allocation

There are a number of factors which may be viewed as barriers to the adoption of dynamic resource allocation systems. Some of these issues are prevalent in existing statically allocated systems, while others are specific to the dynamic context.

2.6.1 Security

Where servers are wholly provisioned the security implications are identical to those of a non-dynamically provisioned server. The servers are exposed to identical threats, and should be considered as such. Where applications are co-hosted on a physical machine there are a number of security considerations to be made. It is important that each application is isolated from each other so that a security issue cannot affect multiple applications. In a virtualised environment, virtual servers are isolated by a hypervisor, which prohibits communication between the physical host and other virtual servers on the host. The security of hypervisors has been extensively studied [38, 60]. Additionally there has been some work on ensuring application security in the cloud [20].

2.6.2 Scalability

Resources can now be allocated on an unprecedented scale which means that the scalability of resource allocation mechanisms is important, as it is a limiting factor in their adoption. The dynamic resource allocation problem is known to be NP-hard (see Chapter 4 for a reduction to the binary integer programming problem), which means that it is often approximated using heuristics. To the best of our knowledge, the work in [28] is the only work which explores this.

2.6.3 Qualify of Service

The performance of a dynamic resource allocation system itself may not be a meaningful metric. The performance of the system should be considered as the improvement it makes the QoS of the applications which it supports. However with this in mind, both the resource allocation algorithm, which decides the resource allocation, and provisioning mechanism, which effects the changes required by the resource allocation algorithm, should be able to devise and effect new resource allocations in a reasonable period of time. There is a large body of work focussing on the performance delivered by specific resource allocation policies [41, 65, 71, 80]. Where comparisons exist it is common for the policies to be compared against trivial policies such as proportional resource allocation or static allocation.

2.6.4 Fault Tolerance

At large scale there is a significant failure rate of components. A dynamic resource allocation system should consider the potential for resource failure to impact on the allocations which it makes. Work in [68] offers a resource provisioning policy which accounts for the failure rates of components in the underlying platform. [39] offers an insight into the reliability of network links in a modern data centre and suggests that overall data centre networks show high reliability, with load balancers being the most fault prone components due

to software errors and that the use of network redundancy is shown to be only 40% effective in reducing the median impact in failure.

2.6.5 Heterogeneity

Large data centres are often comprised of a wide range of hardware in a number of configurations. Operational requirements of applications may dictate which platforms can support each application. Evaluating current commercial IaaS offerings suggests that providers offer virtual machines in a number of specifications which are multiples of the smallest offering. The offerings made are the minimum guaranteed resources, and these may be exceeded if space capacity is available on the physical server. There has been some research [47] into resource allocation in heterogeneous environments.

2.6.6 Resource reallocation

The mechanism to support the actual reallocation of resources is an important consideration. In non-cloud environments deployments may occur via custom scripts or machine re-imaging. Cloud environments rely on virtual machines containing a full software stack for their deployment. As a result, network bandwidth and storage subsystems play a significant role in the performance of the resource reallocation. Work in [46] analyses the impact of a virtual machine migration on data centre efficiency, while work in [77] measures the performance impact to an application of migrating a virtual machine.

2.6.7 Software Licensing

In enterprise systems commercial software is still pervasive, as it provided with a good level of support when compared to open-source offerings. Traditional software licensing models are rigid and may be tied to a specific server or number of servers or licensed annually. Each of these scenarios does not sit well on top of a dynamic platform as the number of servers varies over time. Flexible

licensing models which may be charged in relation to the scale and duration for which the product is used are required to help this issue.

Additional related work specific to each chapter can be found in Sections 3.1, 4.1, 5.1 and 6.1.

2.7 Summary

This chapter has presented the evolution of enterprise application hosting and identified a number of issues which affect the viability of dynamic resource allocation. Much of the work done to date has focussed on the improvements in performance and quality of service. There has been little consideration of issues around the operation of dynamic platforms such as those found in cloud computing and dedicated environments.

This thesis will focus on the effectiveness of dynamic resource allocation at scale, the issue of resource failures and the cost of migration.

CHAPTER 3

An Enterprise Testing Platform

Many different policies for dynamic resource allocation in enterprise systems have been proposed [42, 55, 64]. Much of the work has proven to be theoretical, with policies being evaluated through analytical modelling. The use of analytical modelling may mask various nuances of systems in the real-world, e.g, caching and network latencies, which may in turn have an impact on the potential benefits offered by a policy.

In this chapter a real-world testbed is developed for the purpose of evaluating theoretically derived policies. Three proposed theoretically derived switching policies are implemented and subjected to synthetic workloads and the results of the policies are compared against each other using a static allocation as a reference. The observed behaviour of each policy is evaluated and discussed.

3.1 Specific Related Work

Work by the authors of [50] studies the methods for maximising profits of the best-effort requests and the QoS-demanding requests in a Web farm, however, they assume a static workload arrival rate in the paper. The model they develop is a queueing network model for a two-tiered applications. The experiments presented are executed as simulations with three applications distributed across ten servers.

Work in [32, 72] uses provisioning techniques to achieve Service Level Agreements (SLA). This research uses analytical models to explore system capacity and allocates resources in response to workload changes to obtain guaranteed performance. In [72] resources are not balanced between applications, rather they are added from an idle pool of servers. This ignores the potential competition between resources when the system is near maximum capacity.

Other work in [27, 78] uses admission control schemes to deal with overloading and achieve acceptable application performance. The authors of [27] use session-based admission control to avoid loss of long sessions in Web applications and guarantee QoS of all requests, independent of a session length, while [78] presents a set of techniques for managing overloading in complex, dynamic Internet services and is evaluated using a complex Web-based email service. The work in this chapter focuses on the scenario where multiple applications are running simultaneously in an Internet hosting centre.

The authors of [13] describe the building of a dynamic infrastructure platform prototype. The policies implemented by the platform are based around client SLAs, which are delivered through resource migration. The platform has a notion of free or idle servers which are provisioned wholly for a given application. In the reallocation process described, a full system installation is required when allocating a server which takes between 710 and 750 seconds. This platform differs from the work presented here as the platform is not entirely dynamic; there is a fixed pool of allocated servers and a pool of servers which can be

added or removed from the servicing of an application dynamically.

Work in [45] presents a system for the management of power and performance in cloud infrastructures. The test platform developed uses a three tiered application architecture implemented through virtual servers. In this infrastructure four applications are spread across up to 20 virtual servers, hosted on eight physical machines.

Recent work [42, 55] also studies performance optimisation for multiple applications in Internet service hosting centres, where servers are partitioned into several logical pools and each logical pool serves a specific application. They address the server switching issue by allowing servers to be switched between pools dynamically. These results are provided through simulation and not via a representative test platform.

The work in [55, 64] considers different holding costs for different classes of requests, and tries to minimise the total cost by solving a dynamic programming equation. The authors in [42] define a revenue function and use M/M/n queues to derive performance metrics in both pools and try to maximise the total revenue. The results for these papers are demonstrated analytically.

The work in this chapter is different from [42, 55, 64] in the following respects: an actual testbed is used in our evaluations, and thus (i) the application is not synthetic, (ii) the supporting infrastructure demonstrates the subtleties of a real-world platform, and (iii) the switching policies are implemented, feed actual system parameters, and are subsequently evaluated on synthetic workloads.

3.2 System Overview

In this chapter we consider an environment consisting of two applications which are deployed across a set of servers with each application following the "best possible" architecture as described in [69].

In the case of a single application it is common for the presentation tier to

schedule tasks across a dedicated cluster of application server machines. Strategies for request scheduling in both commercial and open-source products are generally variations on the Weighted Round Robin (WRR) strategy. The WRR approach allows for different proportions of requests to be dispatched to different application servers and, in so doing, allows some degree of support for heterogeneous server environments by allocating a higher proportion of the workload to application servers with more capacity.

Applications that require a state to be maintained throughout a user session present a significant problem for WRR strategies, as multiple requests may not be redirected to the same server. To this end several strategies have been developed to handle this scenario. *Session affinity* ensures that subsequent requests are all processed by the same application server, thus ensuring that state is maintained throughout a user session. Drawbacks to this approach are discussed in [34] and include severe load imbalances across the application cluster due to the unknown duration of a request at the time of dispatching it to the application server, and a lack of session failover due to the single application server providing a single point of failure to the session. It is also possible for the client to store the state of the session, resubmitting it with each request. Using this approach any available application server is able to process the user's request. Similarly the data persistence tier may be used to store session data which also enables all application servers to service all requests, however this comes at the expense of increased database server/cluster utilisation. These approaches are evaluated in [19]. In this chapter user session data is stored on the application server that processes the initial request. Further requests are then forwarded to the same server for processing.

The multiple application environment we consider is captured by Figure 2.1. The diagram represents the architecture for n separate applications. The main difference from the single application architecture is the conceptual view of the *set* of application servers. In our multiple application environment any of the servers available may be allocated to any of the applications either statically

or dynamically. In this chapter we are concerned with the allocation of servers at the application tier. Each application requires a dedicated presentation and data persistence tier.

To avoid the overhead of the switching system compromising the performance of any single component in the system, the switching system is hosted on an additional server.

3.2.1 Server Performance

In [31] it is demonstrated that the throughput of an application server is linked to the number of concurrent users. While a system is under a light load with few concurrent users, the throughput of the server can increase in a near linear fashion as there is little contention for resources. As the number of concurrent users increases, the contention for system resources increases, which in turn causes the rise in throughput to decrease. The point at which the addition of further clients does not result in an increase in throughput is the saturation point, T_{max}.

From this it would follow that for a cluster of n application servers, the maximum theoretical throughput of the cluster would be ΣT_{max} for a heterogeneous cluster. This may be simplified to nT_{max} for a cluster of homogenous servers. These theoretical throughputs are rarely achieved in practice due to the additional overheads of scheduling and redirecting requests across the cluster.

3.3 Server Switching

If we consider that each application hosted across the set of servers provides a service to a business, some of the hosted applications are more important than others in terms of revenue contribution to the service provider.

Most Internet applications are subject to enormous variations in workload demand. During a special event, the visits to some on-line news applications will increase dramatically, the ratio of peak load over light load can therefore be

considerable. System overloading can cause exceptionally long response times for requests or even errors, caused by the timing out of client requests and connections dropped by the overloaded application. At the same time, the throughput of the system would decrease significantly [31].

Therefore, it is desirable to switch servers from a lightly loaded application to a higher loaded application in response to workload change. In such cases, it is important to balance the benefits of moving a server to an application against the negative effects on the reduced pool and the switching cost.

3.3.1 The Switching Process

Several different scenarios for server switching are presented in the literature [42, 64]. In [42] it is proposed that the set of servers are shared amongst a single application, which is partitioned according to different levels of QoS. In this case, the simplest approach to reallocating a server would be to remove it from an entry point serving one request stream, and add it to the entry point for the assigned pool. This negates the need to change the application code servicing the requests, which provides a considerable reduction in switching cost. The switching process for this scenario is given in Algorithm 1.

In Algorithm 1, line 1 iterates over each of the applications. On line 2 S_i is defined to be the change in the number of servers required to service the application, it may be positive or negative. Lines 5-25 loop until the server requirements are satisfied. Lines 6-14 handle the case where additional servers are required by searching other applications where the server requirements are negative. Once an application has been identified its upstream web server is prevented from forwarding any further work to the server (line 9) and the in-flight requests run to completion (line 10). Line 11 then migrates the server to the new application, before it is then added to the available servers for the upstream system on line 12. Lines 16-23 describe the inverse process, finding an application which requires a server and making the appropriate migration.

There is a cost associated with migrating a server from one application to

Algorithm 1 Switching algorithm for single application QoS requirements.

```
1: for Application A_i, in applications A_{1..n} do
2:   Let S_i be the number of servers required for A_i
3:   Let AS_i be an application server belonging to A_i
4:   Let W_i be a Web Server belonging to A_i
5:   while S_i ≠ 0 do
6:     if S_i > 0 then
7:       for A_m in A_{i+1...n} do
8:         if S_m < 0 then
9:           Stop W_m dispatching requests to AS_i
10:          Wait for pending requests to complete
11:          Switch server from A_m to A_i
12:          Allow W_i to dispatch requests to AS_i
13:        end if
14:      end for
15:    else
16:      for A_m in A_{i+1...n} do
17:        if S_m > 0 then
18:          Stop W_i dispatching requests to AS_i
19:          Wait for pending requests to complete
20:          Switch server from A_i to A_m
21:          Allow W_m to dispatch requests to AS_i
22:        end if
23:      end for
24:    end if
25:  end while
26: end for
```

another. The cost of a migration is derived from the duration of the migration, and can be considered as the degradation of the throughput in the environment whilst a server is unable to service requests for any application as it migrates.

3.3.2 Switching Policies

A switching policy is defined as an algorithm that when provided information on the current state of the system makes a decision on moving to another state. When doing this the policy must balance the potential improvement in QoS against the cost of performing the server switch. There are several examples of switching policies in the literature [42, 64]. Some of these policies are executed as a result of each arrival or departure of a request; while others are executed after a fixed time period and use statistics gathered over a time window to

inform the switching decision. A policy may also consider request arrivals as being *on* or *off*, which is dictated by any arrivals in a given time period. The work presented in [64] describes four possible switching policies, three of which are implemented in this chapter:

- The *Average Flow Heuristic* uses information on the arrival and completion rates of requests for each application in order to make a switching decision. This heuristic averages arrivals over the duration of the experiment and does not consider the distinct on/off periods for each application. This action requires that a weighted average arrival rate is calculated; this is shown in Equation 3.1, where λ' is the averaged arrival rate of the arrival rate (λ) in the busy periods (m) and the idle periods (n). Algorithm 2 is then used with the calculated average arrival rates. Line 1 of the algorithm initialises the total costs for each jobs queue to 0. The best decision cost is set to positive infinity on line 3. Both job types are checked to ensure that the completion rates are grater than zero (line 4) and an error is returned if they are not (line 5). The loop on line 7 starts from zero and iterates to the current number of servers allocated to job type 1. Lines 8 and 9 calculate the total costs for each job queue and if the overall cost for the new allocation is lower than the current best decision cost then the new allocation is stored (lines 11-12). Lines 15-22 execute the same steps for the second application. Finally the allocation with the best decision cost is returned on line 23.

  The Total Cost algorithm first evaluates if any jobs exist in the queue (line 1), if no jobs exist on the queue then a total cost of 0 is returned (line 28). Where tasks exist in the queue, it is checked for stability (a completion rate greater than the arrival rate) on line 2. If the queue is unstable then a value of infinity is returned (line 25). An array of capacity to include all servers currently being migrated +1 is created and asigned to *st* on line 3. The number of migrating servers is iterated over and the rate at which

migrating servers complete migration is stored at the relevant position in the array *st* on line 5. The total cost is initialised to 0 on line 8. The rate of queue drainage is calculated at each possible migration rates between lines 12-22, and stops when the length of the queue would drain to zero. Finally the calculated total cost is returned on line 30.

$$\lambda' = \frac{\lambda \times m}{m + n} \tag{3.1}$$

- The *On/Off Heuristic* attempts to consider the "bursty" nature of requests to each application. To do this it classifies each application's requests as being on or off, and switches servers accordingly. To account for the on and off periods in the job streams, the arrival rate is calculated as in Equation 3.2; Algorithms 2 and 3 (explained above) are then used to calculate a new server allocation.

$$\lambda = \begin{cases} \lambda & \text{If the job stream is active.} \\ 0 & \text{Otherwise.} \end{cases} \tag{3.2}$$

- The *Window Heuristic* uses statistics gathered over a sliding window of time to calculate arrival and completion rates for each application within a time window. In so doing, the policy ignores the presence of any off periods in the time window. This algorithm is shown in Algorithm 4. The algorithm attempts to find teh lowest best decision cost by first initialising the best decision cost to be positive infinity on line 2. A number of servers are initially assigned to n_1 proportionally based on the job costs for job 1 (line 3), with the remainder of the servers being allocated to job type 2 (line 4). The algorithm then iterates from 0 to the number of total servers calculating the utilisation of servers in each pool for each allocation on lines 6-7. If the utilisation of each pool is less than 1 the cost of the switch is calculated (line 10) and compared against the current best decision cost on line 11. If a better configuration is found the cost

and allocations are stored on lines 12-14.New server allocations are then defined by comparing the current allocation with the calculated values on lines 18-19. The new allocation is returned on line 20. It should be noted that where the utilisation in either pool is grater than or equal to 1, no migrations occur under this policy.

Algorithm 2 Server allocation algorithm.

Input: Current server allocation S_1, S_2
Arrival Rates, λ_1, λ_2
Completion Rates, μ_1, μ_2
Queue Lengths q_1, q_2
Switches in progress $w_{1,2}, w_{2,1}$
Switch Rate $r_{1,2}, r_{2,1}$
Job costs c_1, c_2
Switch costs $sc_{1,2}, sc_{2,1}$
Total costs for job queues tc_1, tc_2
Output: New server allocation, S_1', S_2'

$tc_1, tc_2 \leftarrow 0$
Let bdc be best decision cost
$bdc \leftarrow \infty$
if $\mu_1 = 0$ and $\mu_2 = 0$ **then**
  return error
end if
for s in S_1 **do**
  $tc_1 \leftarrow$ Call Algorithm 3 with parameters $s, S, \lambda_1, \mu_1, w_{2,1}, r_{2,1}, q_1$
  $tc_2 \leftarrow$ Call Algorithm 3 with parameters $s, S, \lambda_2, \mu_2, w_{1,2}, r_{1,2}, q_2$
  if $(c_1 \times tc_1 + c_2 \times tc_2 + sc_{1,2} \times s) < bdc$ **then**
    $S_1' \leftarrow -s$
    $S_2' \leftarrow s$
  end if
end for
for s in S_2 **do**
  $tc_1 \leftarrow$ Call Algorithm 3 with parameters $s, S, \lambda_1, \mu_1, w_{2,1}, r_{2,1}, q_1$
  $tc_2 \leftarrow$ Call Algorithm 3 with parameters $s, S, \lambda_2, \mu_2, w_{1,2}, r_{1,2}, q_2$
  if $(c_1 \times tc_1 + c_2 \times tc_2 + sc_{2,1} \times s) < bdc$ **then**
    $S_1' \leftarrow s$
    $S_2' \leftarrow -s$
  end if
end for
return S_1', S_2'

3.4 Experimental Platform

Our experimental platform is based on the architecture shown in Figure 2.1. In the presentation tier we use a custom Web server to dispatch requests onto the application servers via round robin scheduling. The Glassfish J2EE application server running on a Java 1.6 JVM was selected for the application runtime environment. The application server was tuned in accordance with the manufacturer's published guidelines to improve performance [66]. The tuning primarily consisted of setting an upper limit for the number of threads for the HTTP connection acceptance and the database connection pools. Glassfish then manages the thread pools itself, adding and removing threads between the minimum and maximum values as required. Additionally the memory limit of the Java runtime was set near to the free memory available on the system. For the data persistence tier the Oracle 10g relational database system was chosen, which is representative of production systems that one might find in the field.

The hardware for the presentation tier consists of two dual Intel Xeon 2.0GHz servers with 2GB of RAM. For the application tier, a server pool of eight homogeneous servers is used. The servers all use dual Intel Xeon 2.0 GHz processors with 2GB RAM installed. They are connected via a 100 Mbps ethernet network. The Web servers for each application were comprised of the same hardware. The database servers were all configured as dual Intel Xeon 3.0Ghz CPU servers with 2GB RAM and were connected to the network via a gigabit ethernet connection.

The application used for the testing of the system was Daytrader [12], an open-source version of IBM's performance benchmark Trade. This application was chosen as it is representative of a high throughput Web application. The work presented in [11] suggests adopting an exponential distribution with a mean of seven seconds as a reasonable "think time" for the trade application. This application is used for both application pools, to reduce the number of variables in the experiment.

To generate dynamic workloads a custom load generation system was devel-

Algorithm 3 Total cost algorithm.

Input: Switched servers s
Server Allocation S
Arrival rate λ
Completion rate μ
Switches in Progress $w_{m,n}$
Switch rate $r_{m,n}$
Queue Length q
Output: Total Cost, tc

1: **if** $q > 0$ **then**
2: **if** $\lambda < S - s + w_{m,n} \times \mu_1$ **then**
3: Let st be an array of size $w_{m,n} + 1$
4: **for** i in $w_{m,n}$ **do**
5: $st_i \leftarrow \frac{1}{(w_{m,n}-i)\times r_{m,n}}$
6: **end for**
7: $st_{w_{m,n}} \leftarrow \infty$
8: $tc_1 = 0$
9: Let vq be the virtual queue length
10: $vq \leftarrow q$
11: **for** j in $w_{m,n} + 1$ **do**
12: **if** $vq > 0$ **then**
13: Let x be the rate at which the queue drains
14: $x \leftarrow vq + (\lambda - (S - s + j) \times \mu) \times st_j$
15: **if** $x \geq 0$ **then**
16: $tc \leftarrow tc + 0.5 \times (vq + x) \times st_j$
17: $vq \leftarrow x$
18: **else**
19: $tc \leftarrow tc + 0.5 \times \frac{-vq}{\lambda-(S-s+j)\times\mu\times vq}$
20: $vq \leftarrow 0$
21: **end if**
22: **end if**
23: **end for**
24: **else**
25: $tc \leftarrow \infty$
26: **end if**
27: **else**
28: $tc \leftarrow 0$
29: **end if**
30: **return** tc

Algorithm 4 Window Policy algorithm.

Input: Current server allocation S_1, S_2;
Arrival Rates, λ_1, λ_2
Completion Rates, μ_1, μ_2
Holding costs, c_1, c_2
Output: New server allocation, S_1', S_2'

Let *bdc* be best decision cost
$bdc \leftarrow \infty$
$n_1 = \frac{(s_1+s_2)\times c_1}{c_1,c_2}$
$n_2 = (s_1 + s_2) - n_1$
for i in $S_1 + S_2$ **do**
  $\rho_1 = \frac{\lambda_1}{i\times\mu_1}$
  $\rho_2 = \frac{\lambda_2}{(S_1+S_2-i)\times\mu_2}$
  if $\rho_1 < 1$ and $\rho_2 < 1$ **then**
    Let c be cost of the switch
    $c = \frac{c_1\times\rho_1}{1-\rho_1} + \frac{c_2\times\rho_2}{1-\rho_2}$
    if $c < bdc$ **then**
      $bdc \leftarrow c$
      $n_1 = i$
      $n_2 = (S_1 + S_2) - i$
    end if
  end if
end for
$S_1' = n_1 - S_1$
$S_2' = n_2 - S_2$
return S_1', S_2'

oped. This allows specified load to be generated for predetermined durations, which allowed us to monitor the reaction of the switching system under repeatable changes in workload. We used two workloads for our experiments. The first workload (shown in Table 3.1) remained static throughout the entire duration of the experiment, and is designed to examine the reactions of the policies. The workload consisted of 1075 concurrent sessions which was derived from the maximum throughout of the whole system. This workload consisted of 1075 simultaneous client sessions, 875 for application 1 and 200 for application 2. Each workload was executed three times.

The second workload was the most dynamic, changing every 20 seconds, which caused the workload to switch within the switching interval. This workload is shown in Table 3.2. Further performance tuning of the application servers increased the total throughput of the platform to 1250 client sessions, which is used as the total number for the workload. Under this workload the 1250 client sessions are distributed across the applications. Further Workload generation was distributed across five slave machines, to avoid the workload generation becoming a bottleneck for the experiment.

To host the switching system, an additional node was added to the architecture in Figure 2.1. This was done to ensure that the additional overheads of the system were not added to any of the existing system components. Although the time taken to switch a server varies, and is in part dependent on the queue of pending requests allocated to the server, we have found that the average time taken to switch a server between pools is approximately 4 seconds[1]. All server migrations occur concurrently. The *switching interval* is the time between executions of the switching policy. In these experiments the switching interval selected was thirty seconds, to allow the possibility for several migration periods within the duration of the experiment.

In the experiments we configure the two applications with different costs to represent the differences in QoS requirements. The *job costs* for our experiments

[1]The range of switching times obtained ranged from 2 to 6.4 seconds.

are considered to be the costs for holding a job. Such a definition allows a value to be attached to a queue of waiting jobs. For our experiments application 1 has a holding cost 25% higher than that of application 2, making jobs for application 1 a higher priority than application 2 as they are more expensive to hold.

Table 3.1: Workload one.

		Timestep							
		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Duration (mins)		1	1	1	1	1	1	1	1
Clients	Application 1	875	875	875	875	875	875	875	875
	Application 2	200	200	200	200	200	200	200	200

Table 3.2: Workload two.

		Timestep					
		T_1	T_2	T_3	T_4	T_5	T_6
Duration (mins)		0:20	0:20	0:20	0:20	0:20	0:20
Clients	Application 1	625	750	375	500	375	550
	Application 2	625	500	875	750	875	700
		T_7	T_8	T_9	T_{10}	T_{11}	T_{12}
Duration (mins)		0:20	0:20	0:20	0:20	0:20	0:20
Clients	Application 1	625	750	375	500	375	550
	Application 2	625	500	875	750	875	700
		T_{13}	T_{14}	T_{15}	T_{16}	T_{17}	T_{18}
Duration (mins)		0:20	0:20	0:20	0:20	0:20	0:20
Clients	Application 1	625	750	375	500	375	550
	Application 2	625	500	875	750	875	700
		T_{19}	T_{20}	T_{21}	T_{22}	T_{23}	T_{24}
Duration (mins)		0:20	0:20	0:20	0:20	0:20	0:20
Clients	Application 1	625	750	375	500	375	550
	Application 2	625	500	875	750	875	700

3.5 Results

The overhead of the system is measured by calculating the maximum throughput of a single server directly, and then measuring the maximum throughput of the server requests that are forwarded from the Web server. We measure

the throughput for each case at a variety of loads as shown in Figure 3.1 using the JMeter HTTP benchmarking tool. It can be observed that the throughput for the system is significantly higher than that of the direct connections. The throughput curves for both connection types fit closely with the typical performance curves seen in [31]. The response time for the direct requests increases dramatically after 100 clients, while the response time for the redirected requests remains constant. The authors believe that this is due to connections between two fixed points (the Web server and the application server) being kept open in a connection pool by the Web server, reducing startup and teardown costs for each connection.

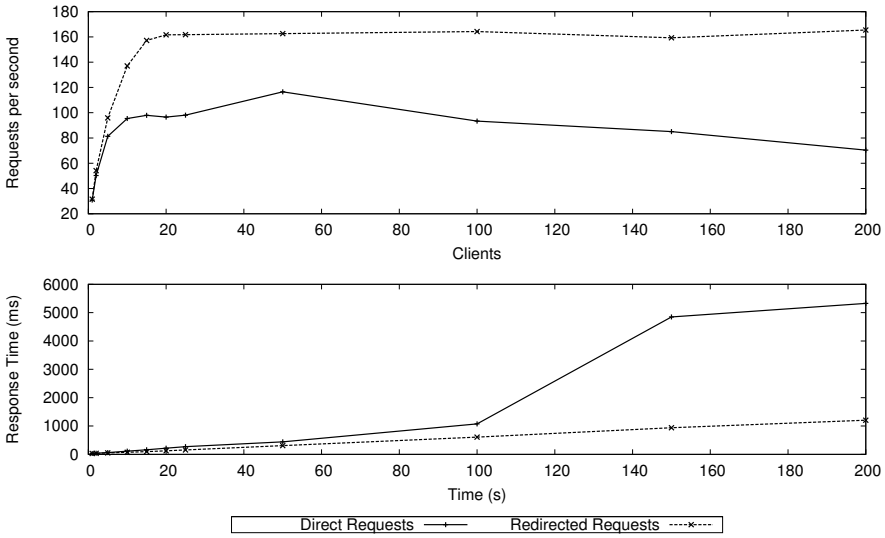


Figure 3.1: Direct server throughput versus redirected throughput.

3.5.1 Static Workload

In this experiment the workload is fixed for each application for the duration of the experiment. The load on each application is shown in Table 3.1.

The baseline for this experiment is provided by using a static allocation of four servers to each application, and measuring the response times under the prescribed workload. The response times for the static allocation are shown in

Figure 3.2. The additional load upon application 1 results in higher response times as the servers are more heavily loaded than the servers for application 2.

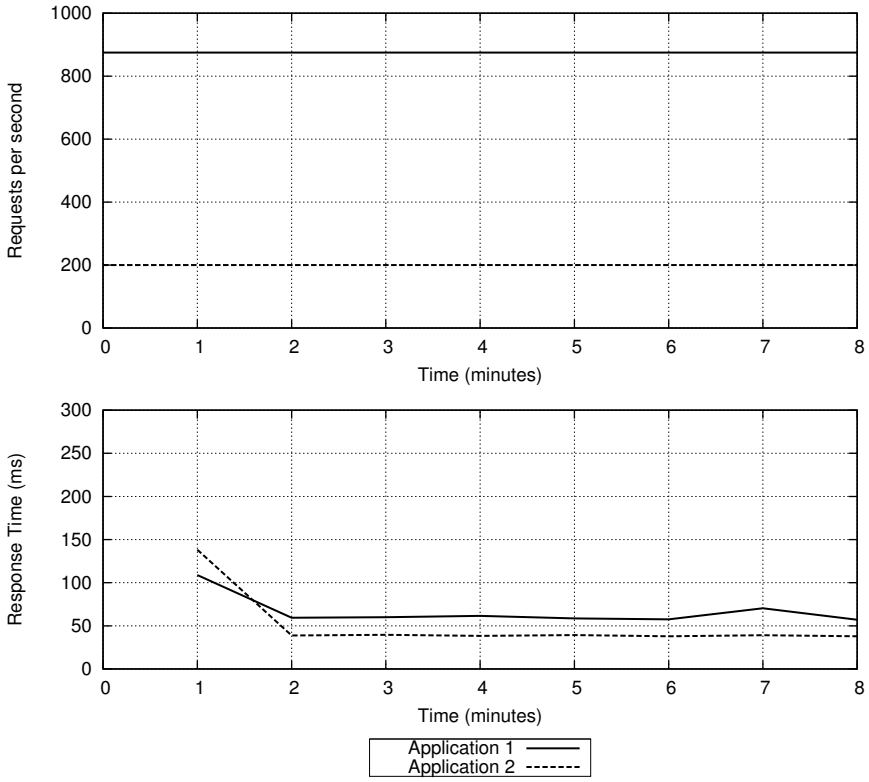


Figure 3.2: Experiment one application response times for a static server allocation.

The initial response times are significantly higher than the latter ones. This is due to the optimisation of the application within the application server, as the Java virtual machine optimises frequently used components. The application server also acts dynamically to improve performance by increasing the number of database connections as required in order to service more requests. In this chapter the application is the same for both pools, so the servers are optimised when they are switched between the pools. As a result we use the first minute as a warm-up period for the servers, and do not use the values in our calculations.

After finding a baseline from the static allocation, each of the three policies were tested against each workload. The results for the three policies for this

workload are given in Figures 3.3 for the Average Flow policy, 3.4 for the On/Off policy and 3.5 for the Window policy . The Figures are set out as follows: the top graph represents the workload for each application. The middle graph shows the server allocation throughout the experiment, and the bottom graph shows the response time for each of the applications. The graphs are aligned such that the x-axis is the same on all three graphs.

The results for the Average Flow policy (Table 3.3) show a 27.38% improvement in response time for application 1 and a decrease of 5.05% for $a2$. Figure 3.3 shows that the policy switched two servers from application 2 to application 1 after two minutes. The difference in throughput (see Table 3.4) is less than 0.4%, and is considered to be a side effect of the stochastic think time used for the clients in the workload.

The On/Off policy results are shown in Figure 3.4. The policy reacted faster than the Average flow policy, switching two servers from application 2 to application 1 after the first switching interval. Although the On/Off policy reacted faster than the Average Flow policy, the response times for application 1 were improved by less. The On/Off policy improved the response time for application 1 by 23.38%. The response time for application 2 was increased by a larger percentage than the Average Flow policy, which is due to the earlier switching of servers.

The Window policy performs the best on the given workload. The results for this policy are shown in Figure 3.5. The policy reduces the average response times for application 1 by 30.20% and increases the response time for application 2 by 7.69%. The window policy performs four switches in the early stages of the experiment before remaining at a steady allocation of six servers for application 1 and two servers for application 2.

The effects on throughput of switching servers between the pools are minimal, and may be considered as side-effects of the distribution of client think times used during the experiment. The throughput of the applications does not increase as the workload does not increase in volume.

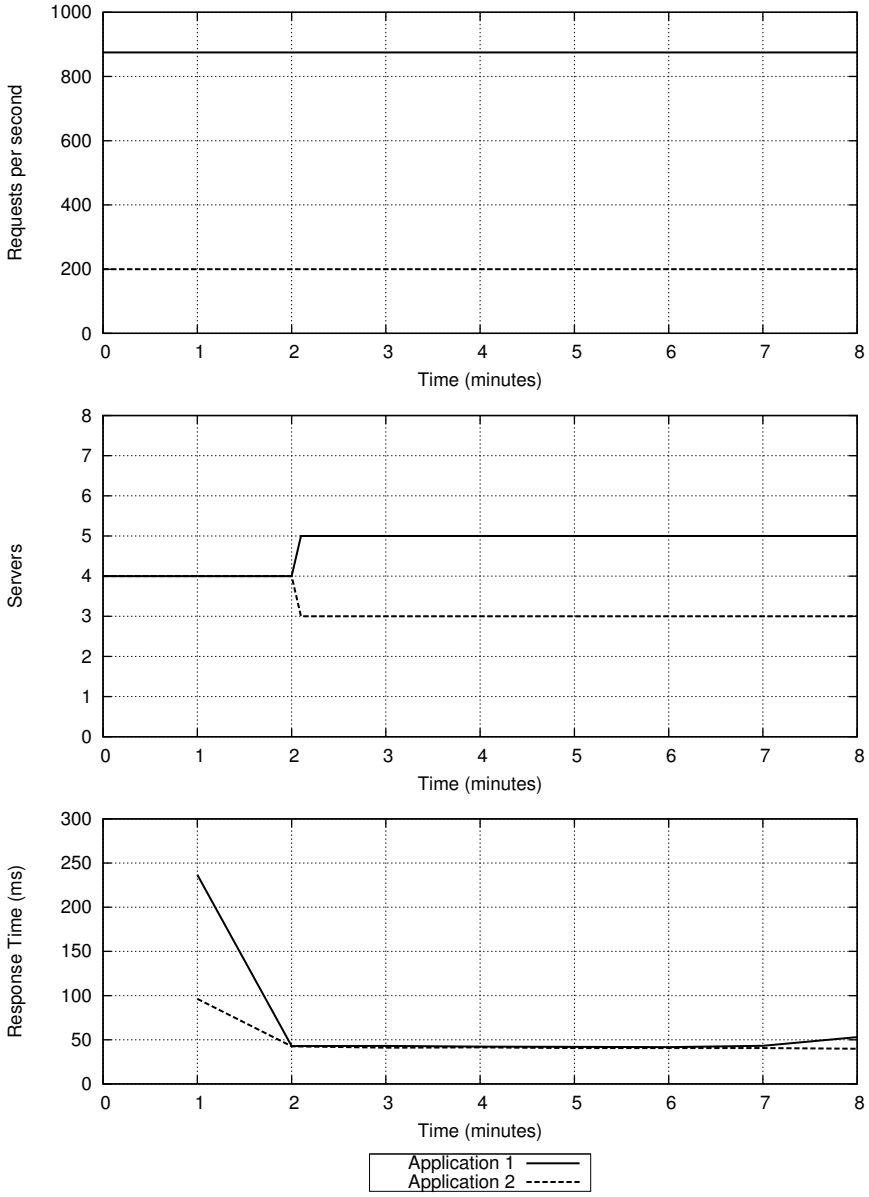


Figure 3.3: Average Flow policy results under workload one.

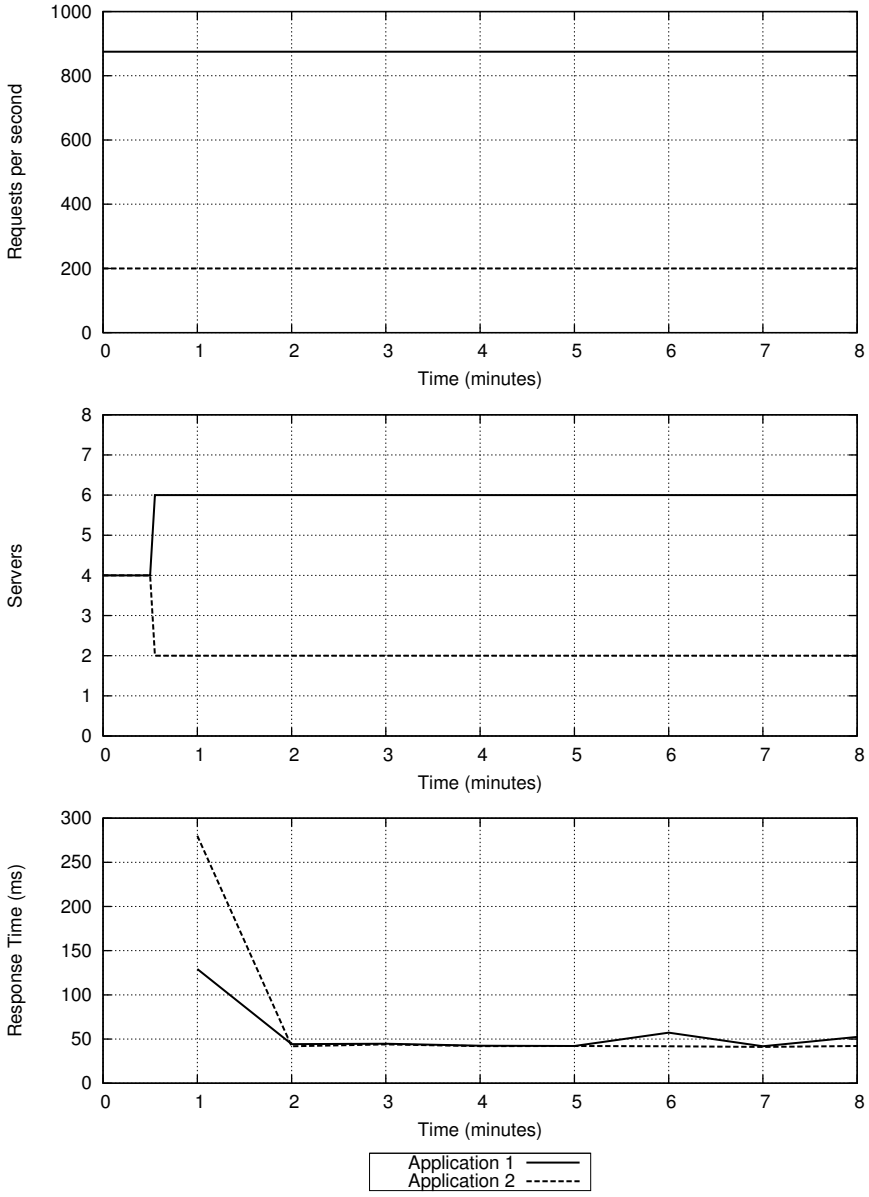


Figure 3.4: On/Off policy results under workload one.

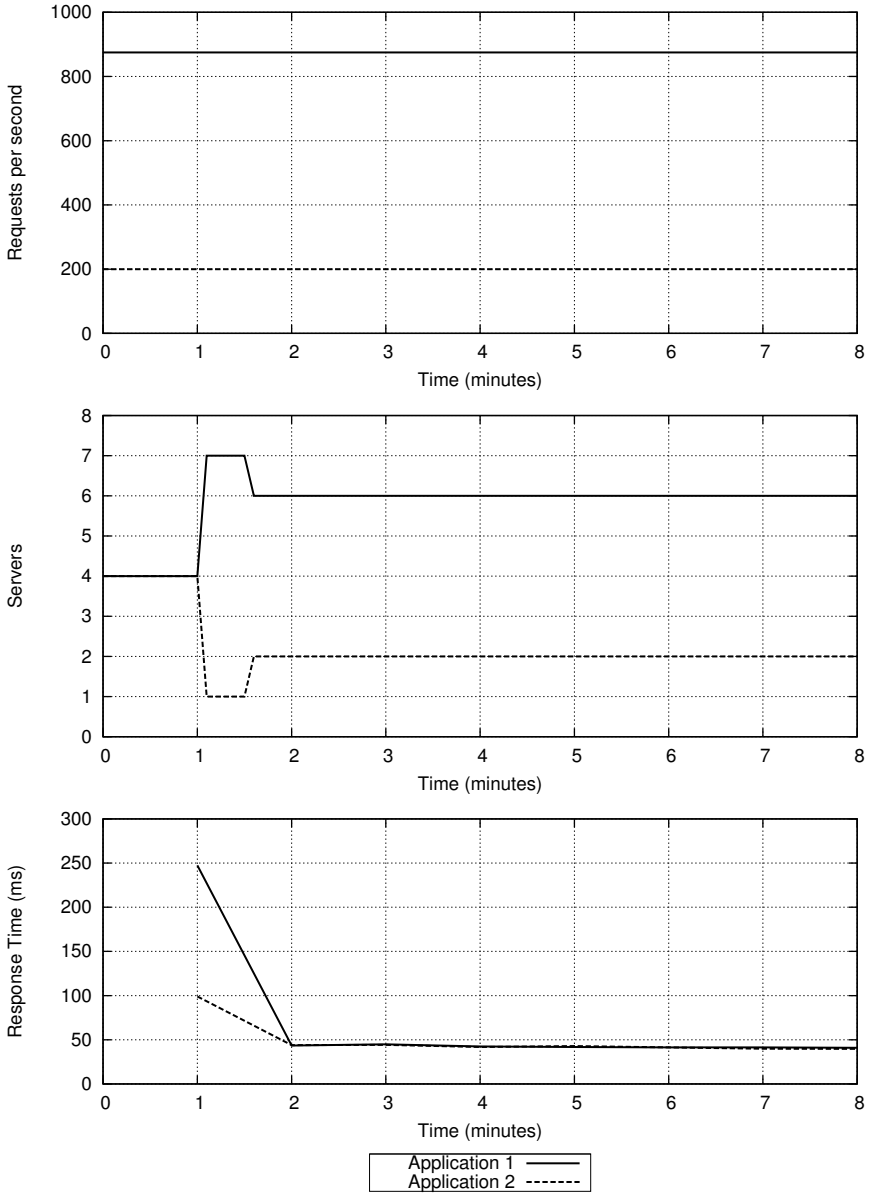


Figure 3.5: Window policy results under workload one.

Table 3.3: Comparison of policy response time against static allocation under workload one.

	Mean response time (ms)			
	Static	Average Flow	On/Off	Window
Application 1	60.60	44.01 (-27.38%)	46.43 (-23.38%)	42.30 (-30.20%)
Application 2	38.99	40.96 (5.05%)	42.40 (8.75%)	41.99 (7.69%)

Table 3.4: Comparison of policy throughput against static allocation under workload one.

	Mean throughput (requests/second)			
	Static	Average Flow	On/Off	Window
Application 1	133.48	133.90 (0.31%)	132.11 (-1.03%)	134.02 (0.40%)
Application 2	28.95	29.02 (0.24%)	28.99 (0.14%)	29.14s (0.66%)

3.5.2 Rapidly Changing Workload

The workload used in experiment two is shown in Table 3.2. The workload changes at twenty second intervals, which is shorter than the switching interval. The baseline for the workload in experiment two was found by observing the performance of a static allocation of servers. The results for the static allocation are shown in Figure 3.6.

The Average Flow policy showed the best performance for this workload, improving the total response time for the applications by 22.84%. The policy performs four server switches over the course of the experiment.

The On/Off policy improved the performance of application 1 by 7.36% and $a2$ by 12.65%. The policy switches the most servers at one interval, switching four servers at four minutes and thirty seconds in the experiment and performs fourteen switches throughout the experiment. The Window policy switches servers in a cyclic pattern, which is synchronised with the workload, with a 2 minute period. The policy does not significantly improve the response time for application 1 (see Table 3.5) but it improves the response time of application 2

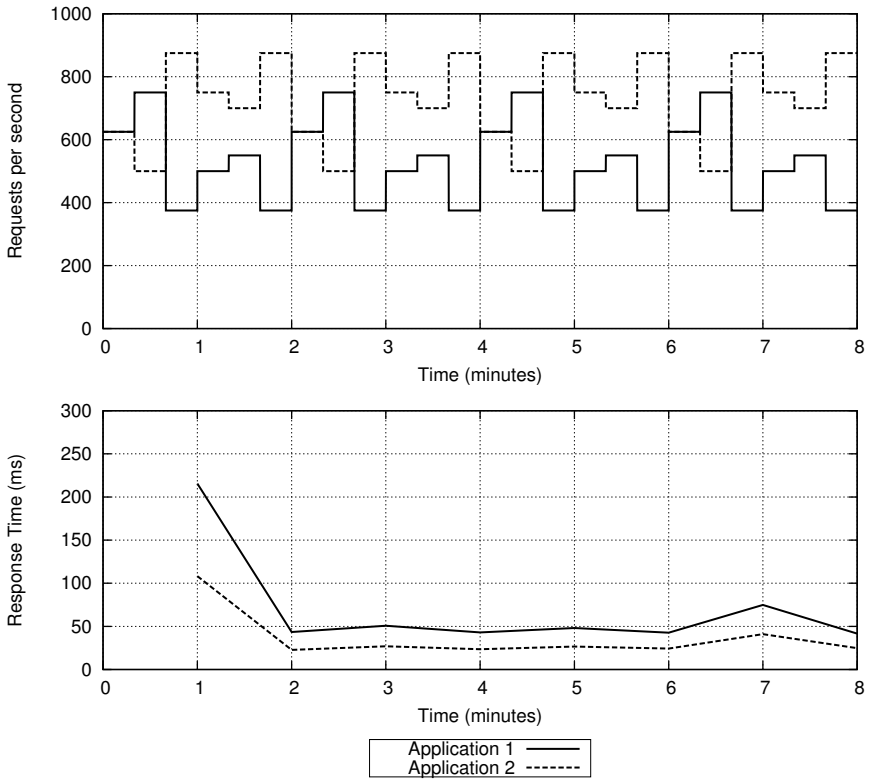


Figure 3.6: Application response times for a static allocation under workload two.

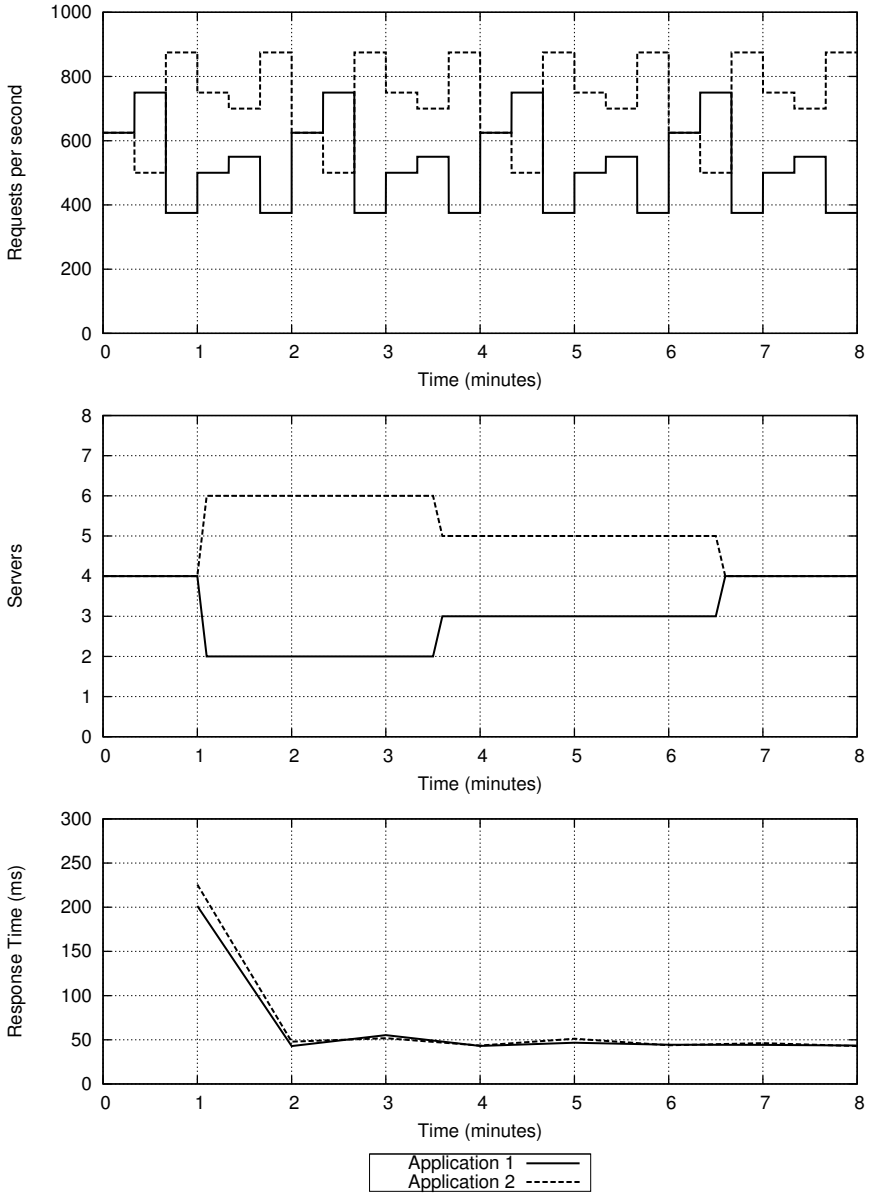


Figure 3.7: Average Flow policy results under workload two.

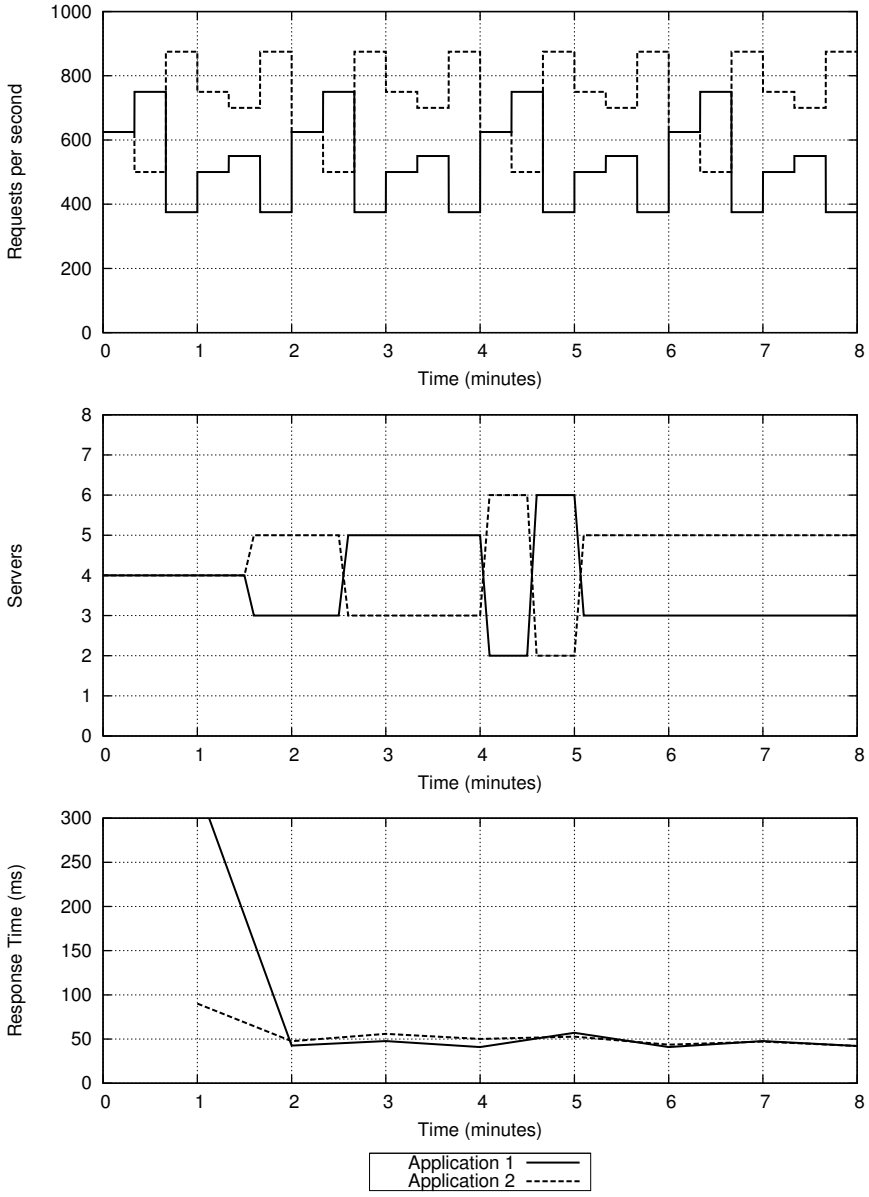


Figure 3.8: On/Off policy results under workload two.

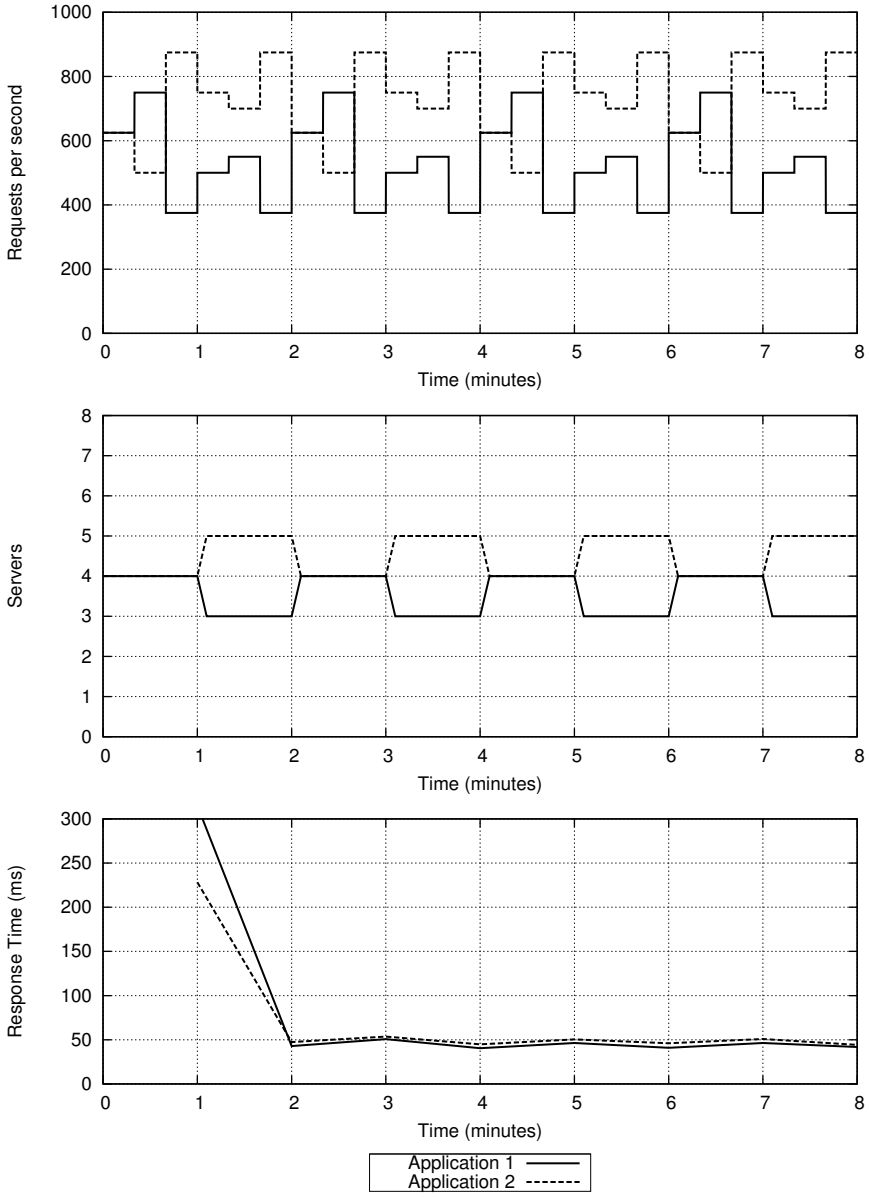


Figure 3.9: Window policy results under workload two.

Table 3.5: Comparison of policy response time against static allocation under workload two.

	Mean response time (ms)			
	Static	Average Flow	On/Off	Window
Application 1	49.21	45.71 (-7.11%)	45.59 (-7.36%)	49.19 (-0.04%)
Application 2	55.48	46.75 (-15.74%)	48.46 (-12.65%)	48.18 (-13.16%)

Table 3.6: Comparison of policy throughput against static allocation under workload two.

	Mean throughput (requests/second)			
	Static	Average Flow	On/Off	Window
Application 1	79.75	79.56 (-0.24%)	79.64 (-0.14%)	81.10 (1.69%)
Application 2	110.73	111.94 (1.09%)	110.77 (0.04%)	111.40 (0.61%)

by 13.16%.

3.6 Analysis

The results for the experiments in this chapter have demonstrated that dynamic resource allocation policies can deliver improves application performance over statically allocated resources. The three policies presented here are taken from the literature, and have been applied to a representative enterprise application testbed.

The Average Flow policy would be well suited to workloads which have relatively stable averages with peaks and troughs of equal magnitudes. Such workloads are common in Web applications, where they experience similar increases in traffic at certain times each day. In this case, the average flow policy would find a relatively static allocation for each application, tending towards an average allocation. This average seeking position makes the policy less suitable for workloads in which the peak is much larger that the average as it would be

slow to respond to these changes.

The On/Off policy has significant potential for bursty workloads which contain idle periods. An example of this type may be order processing in a warehouse, where workloads may be high overnight in preparation for the next day, but then quieter while other work is being done. This may be considered as an online batch processing application.

The Window policy offers the potential for more rapid changes in resource allocations, as it only considers the previous window when deciding on a new allocation. This means that it can handle sudden spikes in workload, however in doing so may incur significant resource migration costs. Where applications are relatively stable for long periods, the policy may make more adjustments than the average flow, as it is sensitive to workload changes in the window period.

Each of the three policies presented here are reactive, basing server allocations on the observed state of the system. While workload forecasting is a hard problem, particularly for Web applications ,it has been used to enhance resource allocation policies in [10]. These policies may benefit from some predictive workload analysis.

3.7 Summary

In this chapter we test three switching policies (Average Flow, On/Off and Window) under two different workload conditions (static and rapidly changing) on a representative test bed. The results show that:

- if the workload remains static for long durations, all policies deliver significant performance improvements for the heavily loaded application (23-30%). In our experiment the Window policy shows the best improvement in response time (30%);
- rapidly changing (with respect to the switching interval) workloads show improvements for both applications under all policies. The Average Flow

policy shows the largest combined decrease in application response times (22%);

- the use of switching policies in these experiments has no significant negative impact upon the throughput of the system. In the worst case the system overhead (reduced throughput) is less than 2% (for the Window policy under workload two).

The results obtained from this platform indicate that the use of switching policies offer the potential for significant improvements in application response times.

CHAPTER 4

Scalability of Dynamic Resource Allocation Policies

Advances in server hardware and software have led to an increase in application density within a modern data centre. It is therefore important that tools and techniques developed to manage these resources can also scale alongside the resources under their control. If the tools and techniques are unable to manage resources at large scale, their utility to large infrastructures is significantly reduced. This area is often overlooked in the presentation of new policies in favour of performance related results.

The greatest limitation of existing dynamic resource allocation algorithms is the limited number of servers and applications that have been used in their evaluation. In this chapter, we directly address this limitation, demonstrating that a class of dynamic resource allocation algorithm, which accounts for many established policies, is not suited to the effective management of resources at data centre scale.

The previous chapter demonstrated the effectiveness of dynamic resource allocation (DRA) policies in a real world testbed at a small scale. In this chapter

we evaluate the scalability of a well-known class of resource allocation algorithm, and propose a new heuristic framework with significantly better scalability characteristics.

4.1 Specific Related Work

Work in [80] utilises a mean value analysis approach to maximising the profit obtained from an enterprise system, with resource allocation across all tiers of a 3-tier architecture. The work is extended in [80] to incorporate an admission control policy which further improves results. The results are provided through simulation, with 25 servers service two applications at the application tier.

In [41] the authors develop an algorithm for profit maximisation of multi-class requests through dynamic resource allocation. The algorithm is developed for two service classes; a best-effort service class and a service class with a guaranteed quality of service (QoS). This work demonstrates a policy which delivers significant improvements in profits, however the algorithm demonstrates poor scalability characteristics as the number of service classes increases.

In contrast, work in [65] develops four heuristic policies for dynamic server reallocation which aims to minimise the holding cost of queued jobs within the system. In [65] the policies are developed for allocating resources across two applications. In the paper the policies were examined through simulation with three servers. Three of the four policies were demonstrated in a small-scale practical context in the previous chapter.

Work in [26] proposed a approach known as cluster-on-demand (COD). This approach is based on dynamic allocation from a common pool to many virtual clusters, each with independently configured environments, access controls and network storage volumes. The experiments associated with this work were performed using a physical cluster composed of 80 servers. This paper serves as a good example of a working dynamic resource allocation scheme, though the number of applications examined is limited.

In contrast to on-demand techniques, [72] explored periodic server provisioning. Here a single application, to which idle servers are added as load increased, was analysed, with short-term and long-term workload prediction being used to guide resource allocation. This work allocates a proportion of the 40 servers available to a single application. The work presented here differs from [72], as the total scale of the environment is fixed, with applications competing for available resources.

This work is differentiated from similar work on dynamic resource allocation algorithm analysis [65] by (i) its focus on scalability, an issue that has not received adequate treatment in existing literature, and (ii) the development and validation of a framework for dynamic resource allocation algorithm design and implementation.

4.2 System Model

The dynamic resource allocation problem is as follows: Given a system with a set $A = \{A_1, \ldots, A_N\}$ of applications, and a set $S = \{S_1, \ldots, S_M\}$ of servers, allocate applications to servers such that overall utility of the system is maximised. Formally, given a set of possible mappings $F = \{F_1, \ldots, F_k\}$ of servers to applications ($F_i : (S \rightarrow A)$), and a corresponding utility function U, find a mapping $F_i \in F$ such that $U(F_i) = max\{U(F_1), \ldots, U(F_k)\}$. In other words, find the allocation of resources that maximises resource utilisation.

The DRA problem can be formalised as follows: Let x_i^j be an allocation of server j to application i. We wish to maximise $U(x_1^1, \ldots, x_N^M)$ subject to the following constraints:

$$x_i^j = \begin{cases} 0, & \text{Server } j \text{ not allocated to application } i. \\ 1 & \text{Otherwise.} \end{cases}$$

$$\sum_{j=1}^{M} \sum_{i=1}^{N} x_i^j \leq M$$

(Applications can not use more than M servers)

$$x_i^j \neq x_k^j, i \neq k$$

(Two applications can not reside on the same server)

Algorithm 5 General class algorithm.

```
1: Let a, ..., z represent the number of servers allocated to each application
2: Let S represent servers available
3: Let B represent best allocation
4: Let V represent the value of the best allocation
5: Init V, B
6: for a = 0; a ≤ S; a++ do
7:    for b=0;∑_a^b ≤ S; b++ do
8:       ⋮
9:       for z=0;∑_a^z ≤ S; z++ do
10:         value = metric(a, b, c...z)
11:         if value > V then
12:            (a, ..., z) → B
13:            value → V
14:         end if
15:      end for
16:   end for
17: end for
18: return B
```

This problem is an instance of binary integer programming problem, which is known to be NP-hard. Thus, exponential complexity is unavoidable. Greedy algorithms are typically used to solve approximations to NP-hard problems. The design of algorithms that approximate the DRA problem is normally of the form shown in Algorithm 5 ([65, 42]). The nested for loops on lines 6-9 iterate through all possible server allocations. The value of the configuration is calculated by some metric on line 10 and then compared with the value of the current best configuration (line 11). If the value of the current configuration is greater that the current best configuration then the configuration is stored (line 12) and the best value updated (line 13). Finally the best configuration is returned on line 18. The complexity of the algorithm being $O(M^N)$.

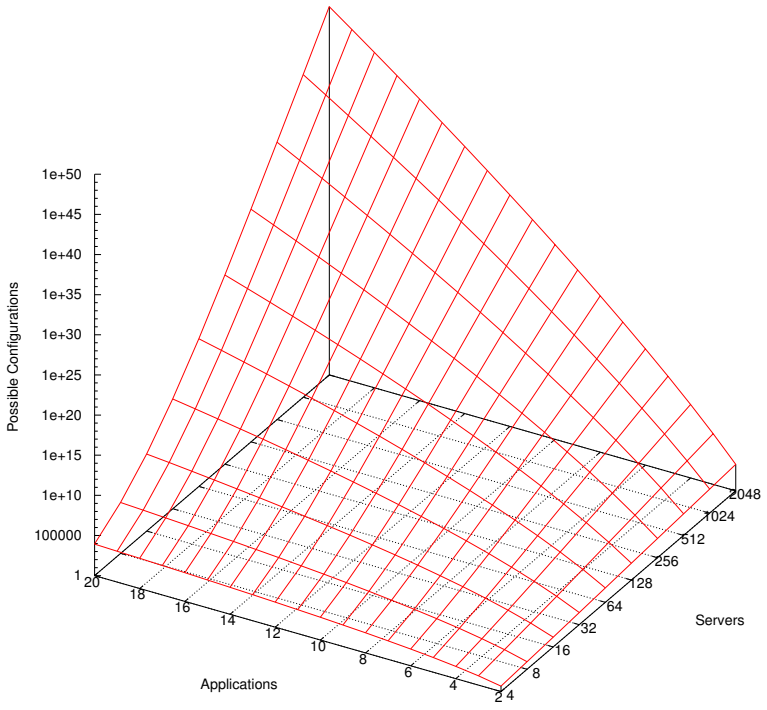


Figure 4.1: Possible server allocations.

4.3 Scalability

To be suitable for data centre scale resource allocation, an algorithm must account for scaling in two dimensions: (i) the number of servers and (ii) the number of applications to be distributed across the servers. Much existing literature has focused on small number of servers, typically with two applications, meaning that little consideration has been given to scalability. However when the number of applications increases, the space of possible combinations increases dramatically. The number of combinations for a given number of servers, s, and applications, a, that wholly allocate all servers can be calculated using $\binom{s+a-1}{s}$. The graph in Figure 4.1 shows the number of combinations for up to 20 applications and 2048 servers.

To understand the rate at which the states could be examined we developed a benchmark using the C programming language to iterate over the number of states. Testing this on a 2GHz AMD Opteron demonstrates that 1.2×10^8

Algorithm 6 Highly Scalable Algorithm.

```
1: Let N be the number of applications
2: Let a_1, ..., a_n be ranked applications
3: Let m_1, ..., m_n be minimum resource of a_i
4: Let t_a be the throughout of a server of a
5: Let n_a be number of servers allocated to a
6: Let I be idle servers
7: for i = 1; i < N; i++ do
8:    p = predictedDemand(a_i)
9:    S'_i = p / t_a
10:   if S'_i < S_i then
11:      Append S_i − S'_i servers to I'
12:      S'_i → S_i
13:   end if
14: end for
15: for i = 1; i < N; i++ do
16:   if S_i ≠ S'_i then
17:      if I' > 0 then
18:         if I' > S'_i − S_i then
19:            Move S'_i − S_i from I' to S_i
20:            break
21:         else
22:            Move all servers in I' to S_i
23:         end if
24:      end if
25:      for j = N; j > i + 1; j−− do
26:         if S'_i − S_i ≤ S_j − m_j then
27:            Move (S'_i − S_i) from S_j to S_i
28:            Break
29:         else
30:            Move (S_j − m_j) from S_j to S_i
31:         end if
32:      end for
33:   end if
34: end for
35: Let I = I'
```

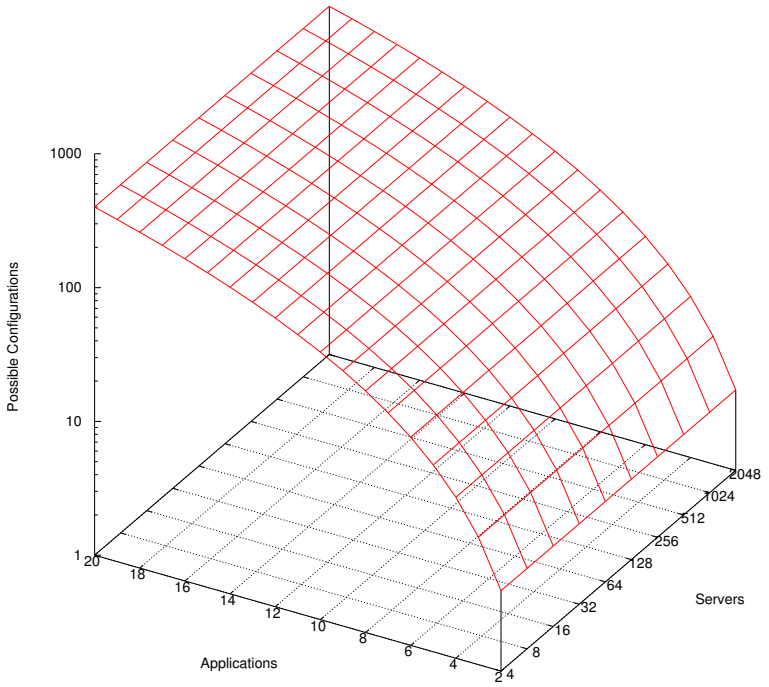


Figure 4.2: Heuristic framework scaling.

states can be iterated over per second, excluding the computation of any metric for each state. The time taken for a DRA algorithm to iterate over all possible allocations and select a configuration forms the lowest bound on the interval between allocations since the calculation for a new state may depend on cost to migrate to the new state from the current state.

The algorithm expressed by the general class may be easily parallelised. To examine the potential speedup an MPI benchmark was developed and executed on a dual processor 16 core AMD Opteron running at 2GHz. The parallelisation was implemented via a partitioning of the outermost loop across all available processors. This offers a good balance of work across all processors with a maximum difference of a single iteration of the outer loop between the largest and smallest amounts of work. The parallelised benchmark was executed three times and the averaged results of this parallelisation are shown in Figure 4.3.

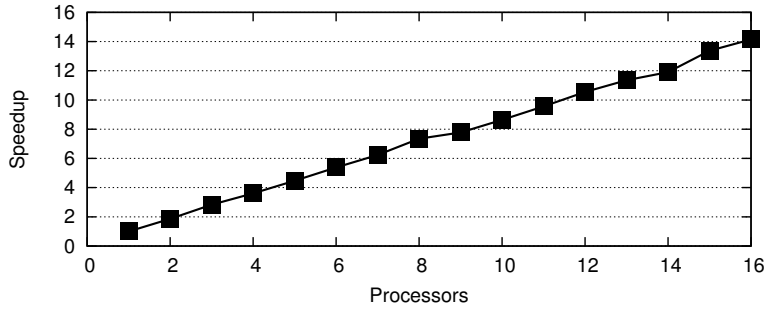


Figure 4.3: State exploration parallelisation speedup.

4.4 A Demand Based Approach

To address the exponential complexity illustrated in Section 4.2, we now propose a template for the development of DRA algorithms suitable for the management of applications and resources at large-scale. The template utilises three distinct phases: (i) the first phase is a preprocessing step done at system setup, (ii) the second phase requires knowledge of throughput of each application, and (iii) the third part runs periodically to reallocate servers to applications.

In the first stage, the applications are ranked in accordance with a predefined metric. We do not provide a specific metric here, as the framework aims to be adaptable, but suitable examples include business criticality, in the context of a single organisation, or *SLAs* in the context of a shared hosting provider. For the second stage, throughput may be measured through performance testing on hardware that is equivalent to that of the deployment target, performance modelling, or measured online at runtime. The accuracy of the performance analysis undertaken will have an impact on the allocation of resources.

Workload prediction for Web applications is a well studied area [40, 81, 9]. Work in [40] characterises demand patterns and uses these to predict future demand for resources, while the authors of [9] apply a number of workload prediction schemes to the allocation policies presented in [80] and demonstrate that considerable increases in profit achieved by the system are possible when workload prediction is enabled.

The periodic stage of the proposed framework is shown in Algorithm 6.

Lines 6-13 reclaim resources from applications which are over provisioned for the next stage. Lines 14-32 allocate resources across applications which are under provisioned for the next period. Lines 16-21 allocate resources from the reclaimed servers if any are available. Lines 23-30 allocate resource from the lower ranked applications to higher ranks, subject to their minimum server allocation. Line 33 returns remaining idle servers. There are several options for servers which remain unallocated after all allocations have been made, and we do not specify any actions here. They may be allocated amongst applications either equally or by some weighted metric. Alternatively they may be switched into a low power mode, if the policy is aiming to minimise power consumption.

Where requests are not serviced within the agreed service level agreement, the hosting provider must pay some penalty. In this chapter we consider the penalty of poor requests to be the inverse of the ranking of the application, i.e. applications of higher order have a higher value to the system and therefore a higher penalty for requests served outside of the SLA.

This framework provides a heuristic to solve the NP-hard allocation problem discussed in Section 4.2. This approach is heuristic-based as (i) it relies on predicted demand to calculate the required resource to be provisioned, and (ii) a ranking of applications to guide resource allocation. We specify a ranking of applications to satisfy the requirements of the most important applications first, thus improving the best-case performance of the algorithm. The worst-case complexity of the framework is $O(N^2)$ when scaling the number of applications. This algorithm is independent of the number of servers. Figure 4.2 shows the scalability of the framework across 20 applications and 2048 servers.

4.5 Experimental Setup

In our experiments we use applications with identical characteristics to limit external parameters. Each application is defined as serving four request types, with service durations of 15, 20, 45 and 110ms ,which represent proportions

0.40, 0.30, 0.25 and 0.05 of the requests respectively. The requests received for each application are distributed by a round robin scheduler to all servers dedicated to the application.

The server reallocation process is comprised of the following steps. Firstly, the round robin scheduler server stops dispatching requests to the server being migrated. Next, the server completes all requests currently queued and the application is stopped on the server. The new application is then deployed to the server. Finally, the server is added to the scheduler and begins to service requests for its new application. The time for the un-deployment and deployment of an application is set at 30 seconds, which is derived from the automated removal and redeployment of Daytrader into a Glassfish application server.

In addition to application performance, the number of failed requests in a system can be an important indicator of overall system performance. This is to say that serving requests quickly at the expense of failures is not desirable. In this chapter we will also use failed requests as a metric to assess the behaviour of each policy. A request is considered to fail if the response time exceeds 60 seconds or it is rejected due full server queues. The queued request limit is set at 10000 requests per server.

To define a policy using the proposed framework we consider the priority of our applications. Applications are ranked by their identifier such that Application 1 is ranked higher than Application 2. We also require resource performance knowledge. For our experiments the throughput of a single server is calculated as 73 requests per/sec in our simulation environment. In order to select a workload prediction algorithm, we have evaluated several in the context of the 1998 World Cup Web traffic data [15]. To select a prediction algorithm we use the 24 hour period from the World Cup Web traffic data which had the highest number of requests. This occurred on 30th June 1998. We examine the following algorithms:

- A last observation approach uses the workload from the previous period as the predicted value for the next period. This works well in situations

Table 4.1: Predictive algorithm results.

Algorithm	Min Pred. Err.	Max Pred. Err.	RMSE
Last observation	333	601104	142574
Sample average	376	2416625	814575
Simple moving average	805	944556	289467
Exp. moving average	553	815174	258210

where the workload does not fluctuate significantly between periods.

- A sample average considers the workload for the next period to be the average of all periods observed. This works well where the workload is constant.

- A simple moving average calculates an average of the previous five periods to make a prediction. Older windows have no bearing on the prediction.

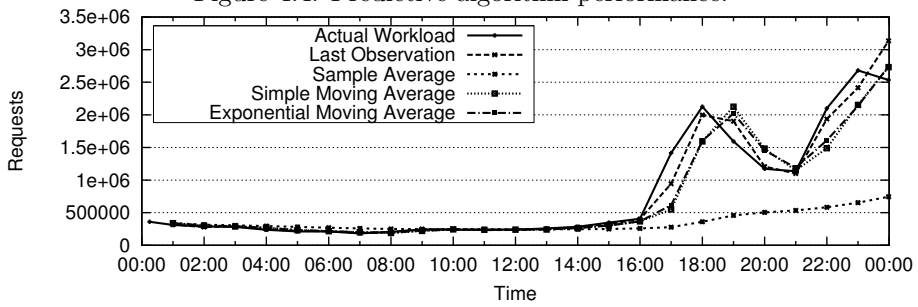
- An exponential moving average is similar to the simple moving average, but it weights the previous periods placing the most weight on the most recent period. In this test we have used five windows.

The predictive performance of these algorithms is shown in Table 4.1. The last observation algorithm yields the lowest absolute error and the lowest root mean squared error (RMSE) in prediction of the tested workload. However, this algorithm is affected by large shifts in application workload. In practice this would mean that the underlying platform may reallocate servers more often. The exponential moving average provides a smoothed prediction of the workload, which allows for clear incremental changes in resource allocation which should remove the potential for significant resource migrations, thus the exponential moving average was chosen as the workload prediction algorithm.

4.5.1 Performance Experiments

To verify the effectiveness of our framework we now compare the performance of the example policy against a policy from the general class and a static resource

Figure 4.4: Predictive algorithm performance.



allocation. The algorithm we have selected for comparison is the Average Flow Policy, which is presented as algorithms 2 and 3 in Chapter 3. To avoid modification of the original algorithm, two applications are used for the performance comparison.

The workload used for the simulations is derived from the log files of the 1998 Football World Cup. The workload selected for each application is an 8 hour section from the 24 hour period. Application 1 is subject to a higher workload than Application 2; 56582546 requests versus 29208765. The time period selected for Application 1 is 16:00 until 24:00 while Application 2 uses requests issued between 12:00 and 20:00. The workload for Application 1 varies between 105 and 3816 requests per second while the workload for Application 2 varies between 202 and 3118 requests per second.

The number of servers used in this experiment is 64, which are initially distributed equally between the two applications. The period between executions of the DRA policies is 15 minutes.

Since the Average Flow Policy allocates all available servers to applications, we do the same for the framework derived policy. Servers which are idle after allocations have been made will be distributed across applications proportionally by rank.

4.5.2 Scalability Experiments

For scalability experiments eight applications were distributed across 64 servers, each of which was subject to a different workload. Applications were initially distributed evenly across servers. Workloads were synthetic, based on sine waves of varying frequency and amplitude, and are shown alongside the respective results in Figures 4.6 and 4.9. The workload durations in these experiments were 8 hours, and the periodic reallocation time is 15 minutes.

4.6 Results

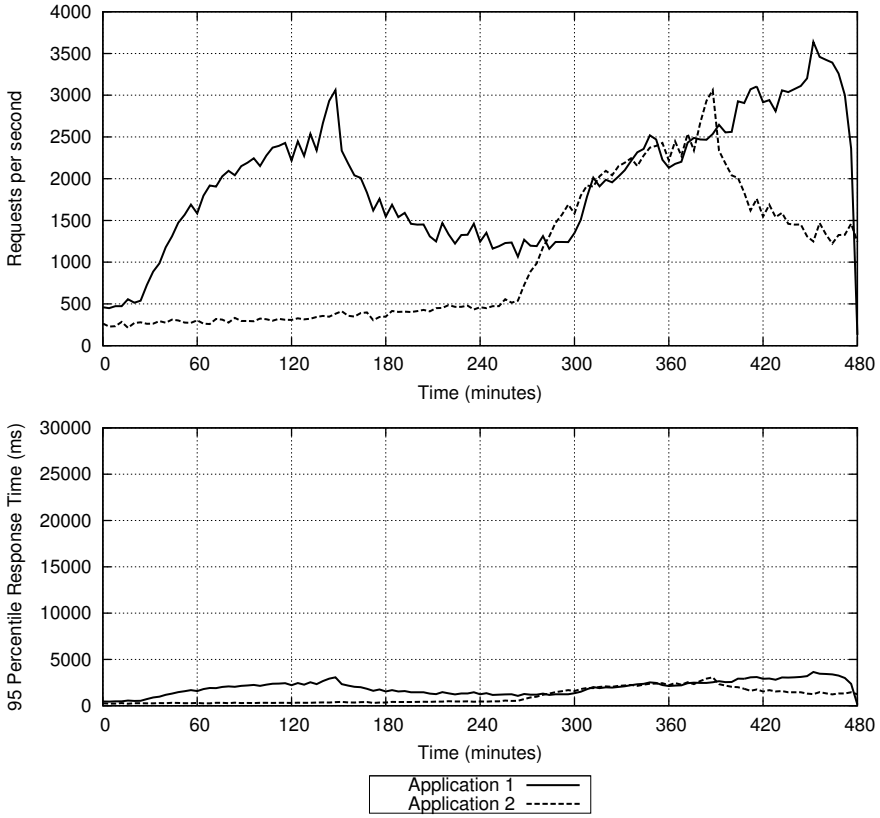
In this section, we compare the performance of the policy derived from the proposed framework against that of the Average Flow Policy and a static allocation. We compare the results based on two criteria; (i) application response times and (ii) the number of requests serviced.

The top graphs in Figures 4.7 and 4.8 show the workload for each application, the middle graph shows the server allocations made for the duration of the experiment and the bottom graph gives the 95th percentile response time for each applications. Figure 4.5 omits the server allocation graph.

From Figure 4.5, the static server allocation performs best, in terms of response time, as its response times never exceed four seconds. The Average Flow Policy demonstrates the best results for the results for Application 1, but this improvement comes at the expense of the response time for Application 2. The response time at 325 minutes exceeds the 60 second timeout for the requests. From Figure 4.8,the framework derived policy delivers the worst response time for Application 1. However, this remains below the timeout threshold for the duration of the experiment. This policy also delivered intermediate results for Application 2, but, again, the response time remained below the threshold. Table 4.2 gives details of the request failures observed by each application under each policy.

The performance of the framework policy is guided by the mechanism used

Figure 4.5: Static allocation results.

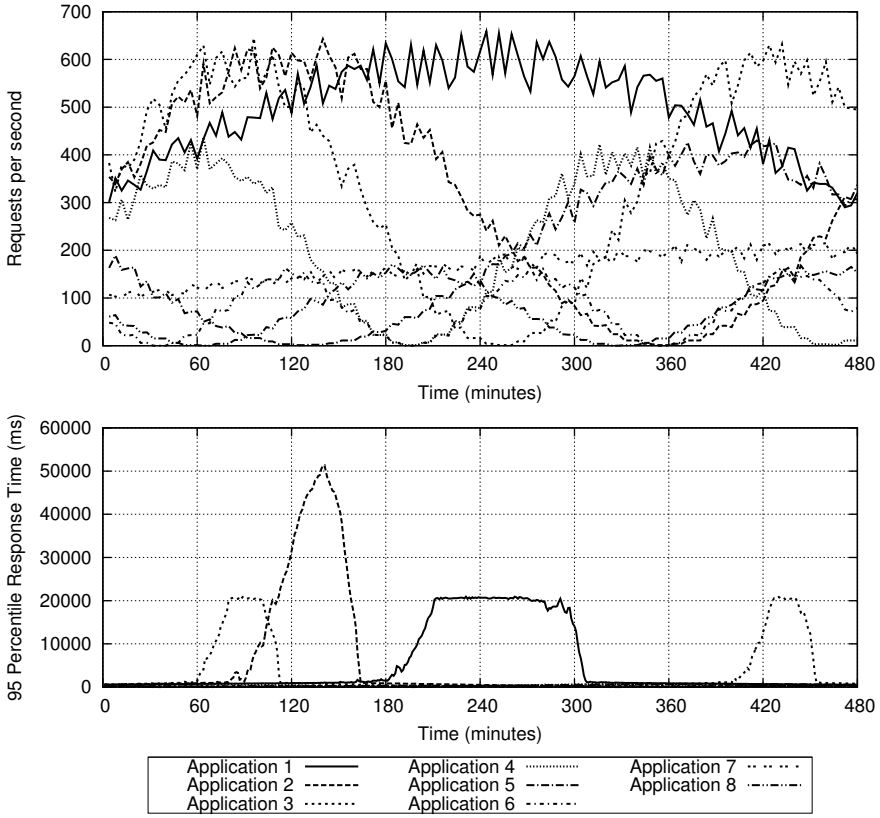


to predict workload. The exponential moving average uses the values observed from the 5 previous windows. Where the workload trend changes, for example Application 1 at 300 minutes, the predication does not capture the increase and the policy migrates servers away from Application 1, which is not the desired behaviour.

The static allocation policy had a failed request rate of 7.03% for Application 1 and 1.24% for Application 2. The Average Flow Policy demonstrates a 0% failure rate for Application 1, but a large 62.73% failure rate for Application 2. The framework policy delivers a failure rate of 2.93% for Application 1 and 5.28% for Application 2. The framework derived policy delivers the lowest overall failure rates from all three of the policies.

In Section 4.5 it was stated that the priority of Application 1 was double

Figure 4.6: Static allocation at scale results.

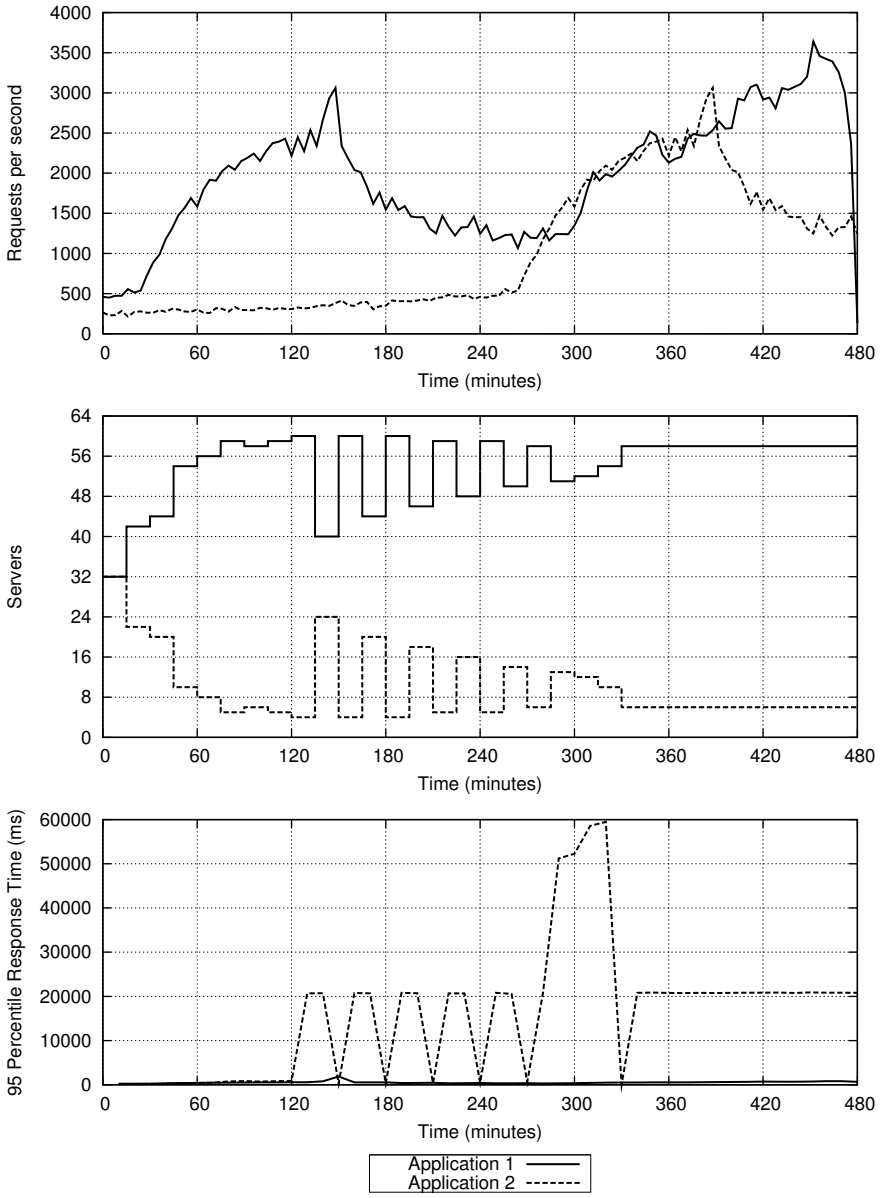


that of Application 2. We use this to infer a cost for each allocation policy by considering request failures for Applications 1 and 2 to have an associated cost of 2 and 1 units respectively. We may then use this metric as a basis for comparison. The static allocation incurs a cost of 8,321,175, the cost of the Average Flow Policy is 18,321,675 and the cost of the framework derived policy is 4,861,395. Hence, the framework policy shows significant cost advantages.

4.6.1 Scalability Results

We now present a larger scale deployment of applications than the performance study. Tables 4.4 and 4.3 show the time taken for the Average Flow Policy and the Framework Policy to make a switching decision. The times for the Average Flow Policy are derived from combining the number of states to be examined

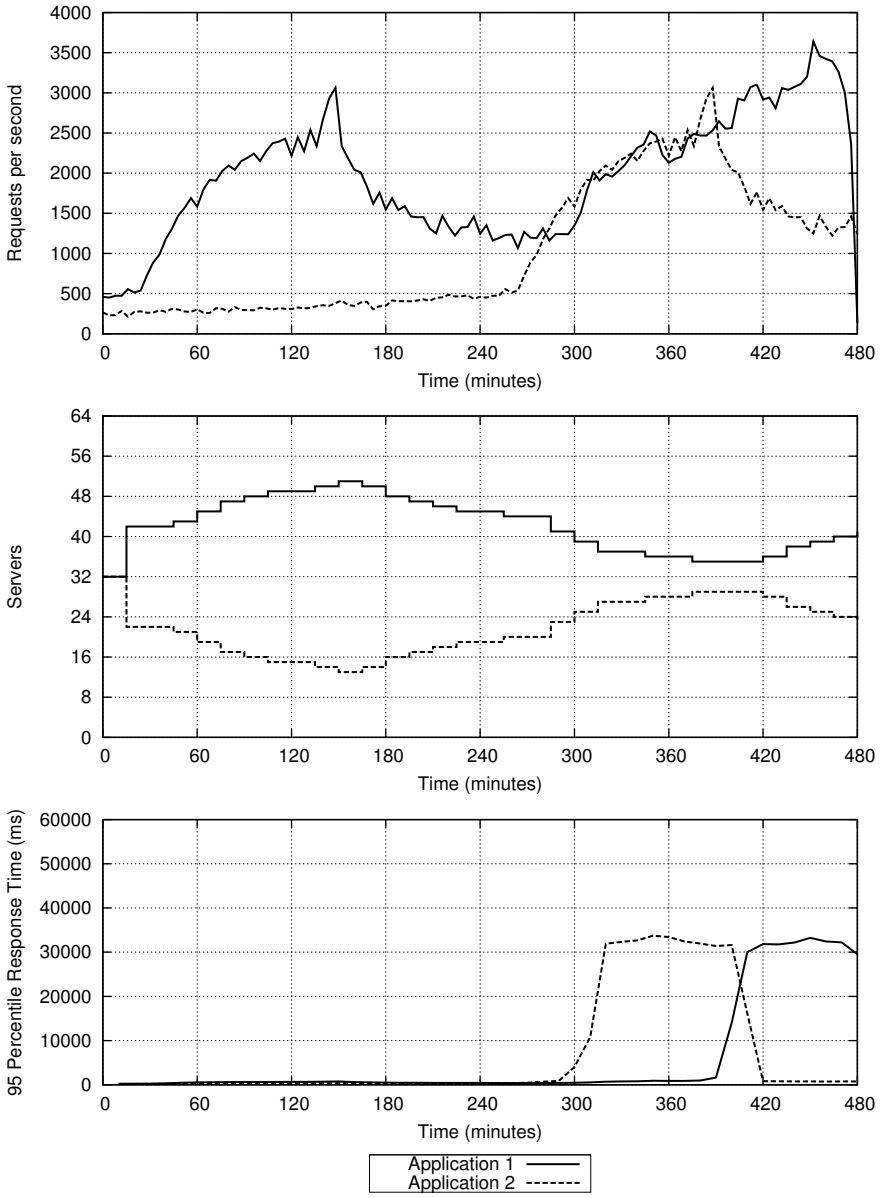
Figure 4.7: Average Flow policy results.



with a timed value for Algorithm 3. A single execution of the algorithm was measured on a 2.0 GHz Intel Core 2 Duo processor and took 1000 nanoseconds in the best case over 20 runs. The times for the Framework policy were obtained from actual executions of the policy.

The Average Flow Policy is currently designed for allocation of servers be-

Figure 4.8: Framework policy results.



tween two applications. To avoid modification of the original algorithm we shall not consider it under the scaling experiment parameters of 8 applications across 64 servers. Table 4.3 shows that under these circumstances the policy would take 22 minutes to develop a new allocation. This exceeds the periodic allocation interval of 15 minutes. Thus the lowest possible bound for server reallocations

Table 4.2: Request failure percentages under performance experiments.

	Requests	Static Alloc.	Average Flow	Framework
App.1	56582546	7.03	0	2.93
App.2	29208765	1.24	62.73	5.28
Total	85791311	5.06	21.36	3.73

Table 4.3: Approximate execution time for Average Flow Policy.

Servers	16	32	64	128	256	512
2 Apps.	1.7e-5s	3.3e-5s	6.5e-5s	1.3e-4s	2.6e-4s	5.1e-4s
4 Apps.	9.7e-3s	6.6e-3s	4.8e-2s	0.4s	3s	22s
6 Apps.	2.0e-2s	0.4s	11s	5 mins	3 hours	4 days
8 Apps.	2.5e-1s	15s	22 mins	38 hours	184 days	61 years
10 Apps.	2.0s	6 mins	1 Day	1 Year	490 years	2.3e5 years

is the decision time of the algorithm. We will exclude the Average Flow Policy from the evaluation of the Framework policy.

The results for the scaling experiment are shown in Figures 4.6 and 4.9. Figure 4.6 shows the performance obtained from a static allocation. The static allocation exhibits more variation in response time than the dynamic policy. The performance of the framework policy is shown in Figure 4.9. It is clear that the dynamic resource allocation is beneficial to the response time of the applications, as the highest 95th percentile result is 20 seconds.

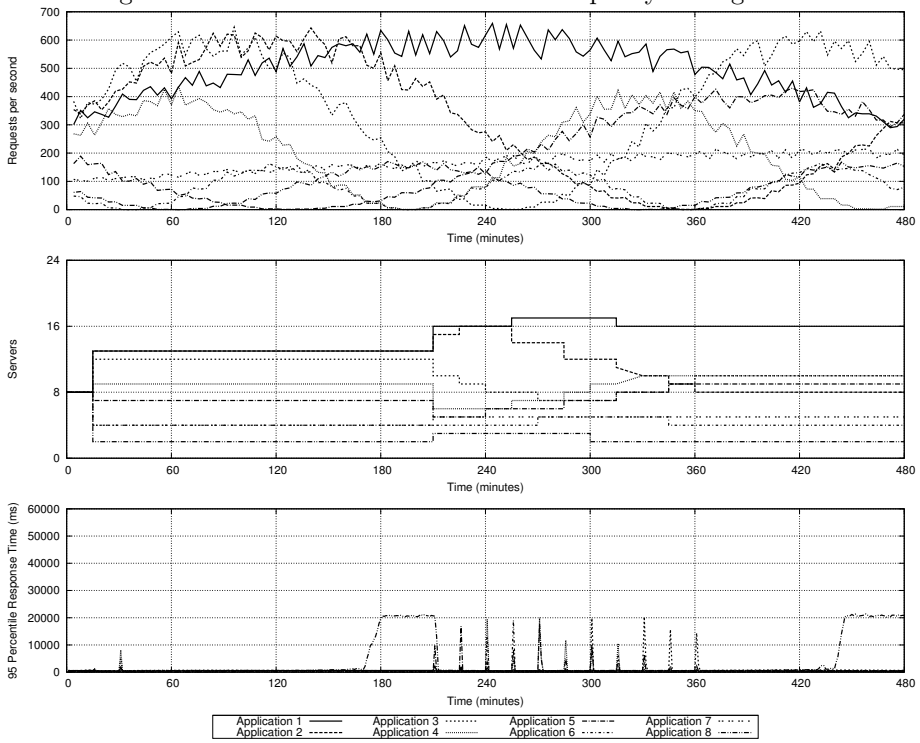
The failures delivered by the system with a large number of applications is reduced by 21% over the static allocation by the derived policy. However, the distribution of failures is also significant. The failure results for both allocation schemes can be seen in Table 4.5. The static server allocation delivers a zero failure rate for Applications 4-8. The failure rate for Application 8 under the Framework Policy is 1.99% which is due to migration of servers from this application to applications with higher priorities. In this case the static allocation outperforms the policy as the application is over provisioned for the duration of the experiment. A statically allocated Application 1 is under provisioned at points in the experiment which causes request failures that do not occur when the policy is used.

The framework policy relies on the applications being ranked by some metric

Table 4.4: Recorded execution time for Framework Policy.

Servers	16	32	64	128	256	512
2 Apps.	5.0e-5 s	6.2e-5 s	5.8e-5 s	5.1e-5 s	5.3e-5 s	5.4e-5 s
4 Apps.	7.7e-5 s	8.9e-5 s	9.1e-5 s	8.9e-5 s	8.9e-5 s	9.8e-5 s
6 Apps.	9.7e-5 s	1.2e-4 s	1.1e-4 s	1.2e-4 s	1.1e-4 s	1.2e-4 s
8 Apps.	1.1e-4 s	1.3e-4 s	1.5e-4 s	1.5e-4 s	1.5e-4 s	1.5e-4 s
10 Apps.	1.4e-4 s	1.7e-4 s	1.9e-4 s	1.8e-5 s	2.1e-4 s	1.9e-4 s

Figure 4.9: Results for framework derived policy at large-scale.



in order to allocate servers to higher ranked applications in favour of those which are of a lower rank. If we now consider that the applications are ranked such that application 1 has the highest priority, with a cost of 8 units for failed requests and application 8 has the lowest priority, with a cost of 1 unit for failed requests we can see that the static allocation costs 440,128 while the policy results cost only 59,181. The policy yields a 7-fold reduction in the cost of failed requests.

Table 4.5: Request failures under scalability experiments.

		Static Policy		Framework Policy	
	Requests	Failed	Failure Rate (%)	Failed	Failure Rate (%)
App.1	14168988	41051	0.29	0	0
App.2	8672845	3192	0.04	0	0
App.3	10490296	14896	0.14	1766	0.02
App.4	6082631	0	0	855	0.01
App.5	5629034	0	0	0	0
App.6	2159599	0	0	259	0.01
App.7	4744868	0	0	0	0
App.8	2186604	0	0	43533	1.99
Total	54134865	59139	0.11	46413	0.09

4.7 Implications and Discussion

The choice of resource provisioning algorithm will vary the benefit obtained for a given workload. Both in terms of its decisions and the frequency at which it can generate new allocations. An algorithm which takes a long period of time time to compute cannot react as quickly as an algorithm that executes faster. This forms the lowest possible bound on the periodic resource reallocation. Thus the scalability of the algorithms in both applications and servers must be considered when evaluating resource allocation policies.

4.8 Summary

In this chapter, we have demonstrated that the DRA problem is NP-hard, and shown that the scalability of a general class of resource allocation algorithms is unsuitable for use in a modern data centre. In response to this we have proposed a framework for the development of DRAs with improved scalability. Further, we have also developed an instance of the framework and compared its performance against that of the general class for a real-world workload to demonstrate the scalability of the proposed resource allocation framework.

CHAPTER 5

Addressing Resource Failure in the Cloud

In order to realise effective and economic models for cloud computing, service providers are turning to large scale, efficient data centres and high speed networks, as well as dynamic resource allocation. To maximise resource utilisation and operational efficiency, and hence profitability, a cloud provider must also make many practical considerations in the design of a data centre, including those relating to physical space, connectivity, power and cooling. These considerations typically make it practical for computational resources to be arranged in collections, hence the necessity of rack based application servicing. However, this arrangement typically leaves collections of resources dependent on common utilities, such as power and network connections, as well as exposing them to a host of co-location issues, such as physical impact and disconnection. This environment is particularly consistent with the notion of a cloud provider offering *Infrastructure as a Service* [73], where an elastic computational resource is leased to a organisation, who must then configure the resource for use. Under this model an organisation would typically be billed for the resource utilised,

thus providing flexibility in situations where workloads are volatile.

Given the scale at which physical resources exist in these environments, failures will be the norm rather than the exception [44]. A single resource failure in a large infrastructure may have limited impact on the system as a whole. In a homogeneous environment the reduction of total system capacity is $\frac{1}{N}$, where N is the number of servers in the infrastructure. The impact to a single application may be significant if there are only a few replicas of the service available, and the excess capacity across all replicas is less than $\frac{1}{A}$ where A is the number of application instances.

In Google's data centres, servers are organised into clusters of 1800 servers, housed in racks of 40-80 servers. In [33] it is suggested that in the first year of operation the following reliability issues will occur:

- One thousand individual machine failures will occur,
- Thousands of hard disks will fail,
- A power distribution unit will fail causing the failure of 500-1000 machines for six hours
- Twenty racks will fail, each removing 40-80 machines from the network
- Five racks will "go wonky" leading to half of the network packets going missing
- There is a 50% chance that the cluster will overheat, taking down most of the servers is less than 5 minutes. This can take 1 to 2 days to recover from.

However, when problems such as power outage or network switch failures occur, a whole rack can fail, leading to multiple servers being unavailable. Failures at such a massive scale are very likely to significantly reduce the efficiency and hence profitability, of the system. Thus, two possible ways of addressing this problem of multiple servers being unresponsive are:

1. Design polices which are failure aware, such that they might make considerations for resources failures in allocating resources;

2. Augment existing policies to make them more resistant to resource failures. The benefit of this approach is that current policies may continue to be used.

3. Execute the DRA policy when a failure is encountered. This has the lowest overhead where there are no failures, however where failures occur re-executing the policy may incur significant costs.

For the design of failure-aware DRA policies, it is unrealistic to expect organisations to modify their policies to capture failure awareness. Rather, it would be better if a failure-aware module can be designed that can work with any DRA currently being used. In this chapter, we propose a modular architecture for failure-aware DRA, whereby a failure-aware allocator module works in tandem with a DRA module. The failure-aware allocator module ensures that resource allocation are made in such a way so as to minimise the impact of rack failures on running applications. The novel property of our approach is that the system has very little overhead as failure-aware allocator only performs some "robustness" balancing of applications, without any additional migrations required to rebalance at any point.

5.1 Specific Related Work

In previous chapters we have presented the performance improvements available through the use of dynamic resource allocation. However the environments in which DRA policies are tested are free of resource failures. Work in [22] offers an approach for virtual machine placement in a cloud for the purpose of minimising the round trip time of a request by placing the server closer to clients in a geographic sense and in doing so potentially improves the robustness of the application through wider geographic distribution of resources.

Resource failures in a data center are a well covered topic in the literature [56, 76, 79]. The coverage of resource failures has mostly concerned failures affecting a single hardware resource, excluding the possibility of large scale failures. The authors of [76] give an insight into the failure rates of servers in a large datacenter, and attempt to classify them using a range of criteria. Work in [44] develops a cloud ready platform for testing a range of failure scenarios, including rack based failures. This demonstrates the need for systems which are able to mitigate against large-scale failures. The issue of resource failures in cloud computing is addressed in [68], where the authors develop a policy to partition a resource between high-performance computing application and Web service applications in the context of single resource failures. The work in this paper increases the scale of the resource failure and in doing so increases the number of applications affected by the failure.

The issue of rack-awareness has been considered to some extent by [63], part of which is a file system, known as HDFS, that can account for rack distribution when storing data. Our work differs from [63] as our algorithm works in conjunction with a DRA system at the level of *Infrastructure as a Service* while Hadoop operates at the level of *Platform as a Service*. Additionally Hadoop offers no support for dynamic reorganisation of data stored under HDFS as new resources are added, whereas our platform distributes across all racks at all times.

5.2 Models

In this section, we present the system and fault models adopted in this paper as well as enunciating our assumptions.

5.2.1 System Model

We consider an environment where a set of applications $A = \{a_1, a_2, ..., a_n\}$ are deployed across a set of racks $R = \{r_1, r_2, ..., r_m\}$. A rack r_i consists of a set of

servers $S_i = \{s_{i,1}, s_{i,2}, ..., s_{i,k}\}$. A server $s_{i,j}$ may service requests from only one application at any time. We assume that (i) all servers $s_{i,j}$ are homogeneous in that they provide identical resources, and (ii) each server has all the resources required by an application, i.e., no communication between servers is required for application servicing. Such a system model is typical in cloud computing environment or large scale datacenters.

5.2.2 Fault Model

Given the described system model, failures are expected to be the norm rather than the exception [44] due to the very large number of computing hardware present. We consider crash failures to occur at a rack level, where this type of failure may be caused by, for example, power outages or network switch failures. When such a rack r_i fails, all the servers $s_{i,j} \in S_i$ become unavailable. We assume a failure mode where at most a single rack can fail at any one time.

5.3 Metric for Failure-Awareness

Well designed enterprise systems can allow for increased throughput via horizontal scaling of resources with minimal overhead [43]. The system may scale linearly with the addition of resources which do not overload any of the other tiers. In a homogeneous server environment of n servers, each server contributes $\frac{1}{n}$ of the total system capacity.

A rack failure will impact on multiple applications depending on the composition of the rack which is affected. To meet performance requirements, resource allocation will have to take place. However, the resource allocation needs to be cognisant of the failures. For example, in HDFS [63], an application will be located on at least two different racks, so that failure of one rack does not cause the application to become unavailable. To assess the impact of the failure of rack $r_i \in R$ on a given application $a_j \in A$, we use the proportion of servers hosting application a_j that will be lost due to the failure of the rack r_i to mea-

sure the loss of application capacity for a_j, and base reallocation decisions on that proportion.

The metric we propose, called *capacity loss*, is shown as equation 5.1, where r_i is the rack, a_j is an application and s_i^j is the number of servers in rack r_i hosting application a_j, and S_{a_j} is the number of servers hosting application across all racks, is used to capture the impact the failure of rack r_i will have on the application a_j in terms of capacity.

$$CapacityLoss_{r_i,a_j} = \frac{s_i^j}{S_{a_j}} \tag{5.1}$$

It is clear that a capacity loss of 1 is undesirable, as it means that all the servers hosting application a_j are located in rack r_i. In an ideal, situation, the capacity loss would be 0. Thus, an objective is the balanced minimisation of the capacity loss metric for all applications across all racks.

5.4 A Modular Architecture for Failure-Aware Resource Allocation

In this section, we present a modular architecture, in Figure 5.1, for failure-aware resource allocation in the presence of rack failures.

In Figure 5.1, the DRA component represents the dynamic resource allocation algorithm that is used by the cloud computing environment. The DRA component takes the forecasted application workload as input, and outputs a set of resource allocation transformation for performance reasons. For example, such an output will be of the following type:

- Take 4 servers from application 4 to host application 2.
- Take 2 servers from application 1 to host application 3.

However, such DRAs are not rack-aware. Specifically, DRA outputs do not specify the racks where the resource transformation is to take place. In

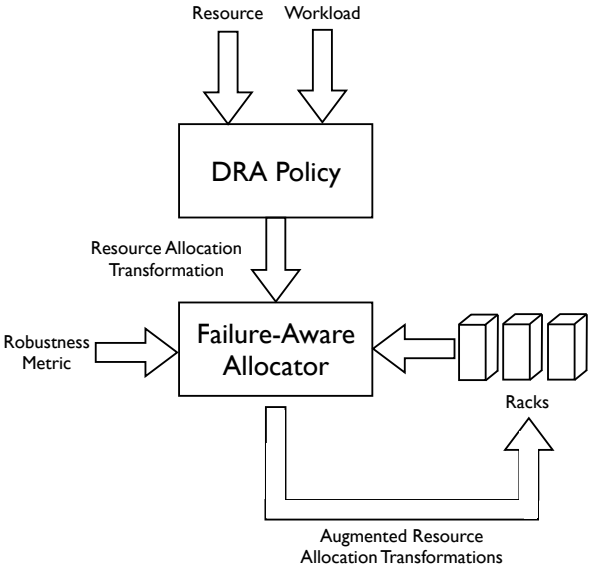


Figure 5.1: A modular architecture for failure-aware resource allocation.

this chapter, one of our contributions is to develop a component, namely the *Allocator* (see Figure 5.1), that performs the rack-aware resource transformation in that it augments the resource transformation needed with rack information, i.e., it specified the racks where the transformations are to occur. For example, with an allocator module, our previous example becomes:

- Take 4 servers from application 4 to host application 2 (2 servers from rack 1 and 2 servers from rack 3).
- Take 2 servers from application 1 to host application 3 (1 server from rack 2 and 1 server from rack 2).

However, with the non-zero probability of rack failures, the allocator needs to be failure-aware in that it needs to perform these resource transformation in such a way to minimise the capacity loss (see Equation 5.1) for each application. The algorithm for the failure-aware allocator is shown in Algorithm 7. In a nutshell, Algorithm 7 searches for all racks where application a_f is being hosted. Then, the failure-aware allocator chooses the set of racks, denoted R_f^t, where the capacity loss for application a_f on a rack r_i (where $r_i \in R_f^t$ is maximum)

and capacity loss for application a_t on r_i is minimum. Practically, this implies that it is better to remove servers for application a_f from a rack $r_i$where the capacity loss for a_f is already high. Further, it also means that it is better to allocate servers to host application a_t on rack r_i where application a_t already has minimum capacity loss. Once these racks R_f^t are identified, one of them is picked at random.

Algorithm 7 Balanced migration algorithm.

Input: a_t The application to be migrated to
Input: a_f The application to be migrated from
Input: R the set of all racks
Output: The optimal rack for migration

1: Let $c_1 = \emptyset$
2: Let $c_2 = \emptyset$
3: Let $c_3 = \emptyset$
4: Let o be an optimal rack
5: Let m_t represent the best value for a_t
6: Let m_f represent the best value for a_f
7: Initialise $m_t = +\infty$
8: Initialise $m_f = -\infty$
9: **for all** $r \in R$ **do**
10: **if** r contains a_f **then**
11: $c_1 \leftarrow r$
12: **end if**
13: **end for**
14: **for all** $r \in c_1$ **do**
15: $v_t = CapacityLoss_{r,a_t}$
16: **if** $v_t \leq m_t$ **then**
17: **if** $v_t < m_t$ **then**
18: $c_2 = \emptyset$
19: **end if**
20: $c_2 \leftarrow r$
21: $m_t = v_t$
22: **end if**
23: **end for**
24: **for all** $r \in c_2$ **do**
25: $v_f = CapacityLoss_{r,a_f}$
26: **if** $v_f \geq m_f$ **then**
27: **if** $v_f > m_f$ **then**
28: $c_3 = \emptyset$
29: **end if**
30: $c_3 \leftarrow r$
31: $m_f = v_f$
32: **end if**
33: **end for**
34: **return** Random member of c_3

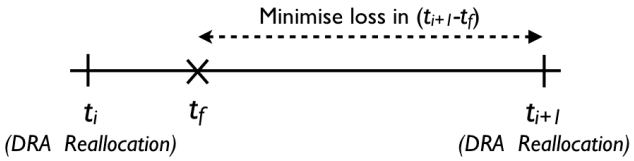


Figure 5.2: Timeline of events.

In the next section, we conduct experiments to evaluate the efficiency of our proposed architecture and failure-aware allocator algorithm.

5.5 Experimental Setup

5.5.1 Failure Scenario

In our experimentation, we considered a DRA policy that executes periodically. Figure 5.2 describes a typical series of events in our experimental environment. At t_i, the DRA policy makes a decision based on the application workloads and returns a set of resource transformations (see example in Section 5.3) to meet the performance requirements. This set of transformations is taken as the input to the failure-aware allocator component, which then decides on which racks to enable these transformations. Specifically, the failure-aware resource allocator module augments the resource transformations with rack location information. A rack failure t_f occurs such that $t_i < t_f < t_{i+1}$, where t_{i+1} is the period for the next DRA decision. We do not consider the possibility of a server failure during a migration in this work.

Thus, with the proposed modular failure-aware DRA framework, we expect the system to better handle rack failures. To do this, we measure the number of failed requests between t_i and t_{i+1}. It is the aim of the research presented in this paper to minimise the system performance loss incurred in the period between t_f and t_{i+1} through rack-awareness in the resource allocation mechanism.

5.5.2 Resource Allocation

Much current research in DRA policies does not explicitly consider resource failure. In such works, each application is viewed as a logical collection of servers which are equal. In this case, servers may be migrated between applications arbitrarily. We use this naïve approach as our benchmark, selecting servers to be migrated as required with no regard for their location. We refer to this allocation mechanism as the *random allocator*.

Under the balanced resource allocator, which uses the $CapacityLoss$ metric shown in Equation 5.1 to assign applications, we attempted to minimise the potential capacity loss in the event of a rack failure. In Section 5.3, we decoupled the failure-aware allocation mechanism, in the allocator, from an abstract DRA policy. For our experiments we employed this approach and used the resource allocation algorithm shown in Algorithm 6 (Chapter 4). The algorithm uses workload prediction and applications which are ranked by criticality (this may be governed by SLA or a business metric) to partition resources.

To reduce experimental uncertainty we used 10 identical applications. Each application processes a single type of request with a fixed service duration of 100ms.

The simulated datacenter was comprised of 400 homogeneous servers that are housed in racks. Each rack contains 40 servers, giving a total of 10 racks. Initially the servers were allocated evenly between applications, i.e., 40 servers per application, with 4 servers per application per rack. The initial allocation of servers to racks was done such that the minimum capacity loss for each application, i.e., maximum robustness for each application, was achieved. When a server was reallocated, it first had to complete the servicing of current and queued requests, before migrating to the newly assigned application. The process of server migration was fixed at 30 seconds.

As the number of applications is high, we use synthetic sine-based workloads of various frequencies and amplitudes. In all cases the total workload of the system is greater than 75% of the total system capacity. The individual workloads

are shown in Figure 5.3, and the total system utilisation is shown in Figure 5.4.

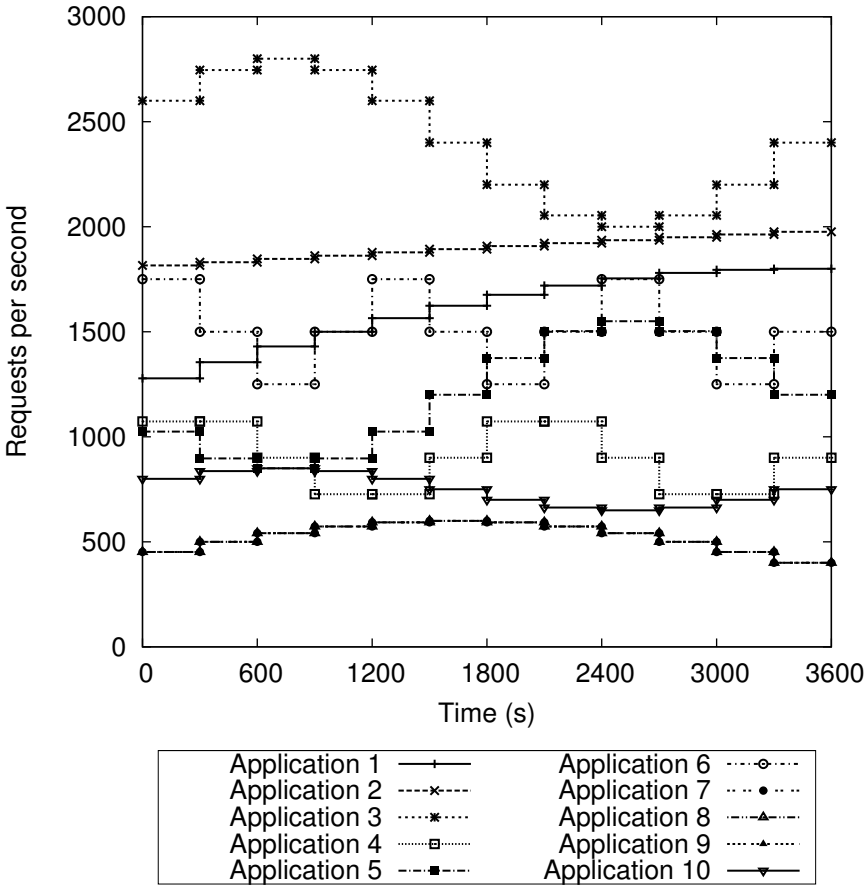


Figure 5.3: Experiment workloads.

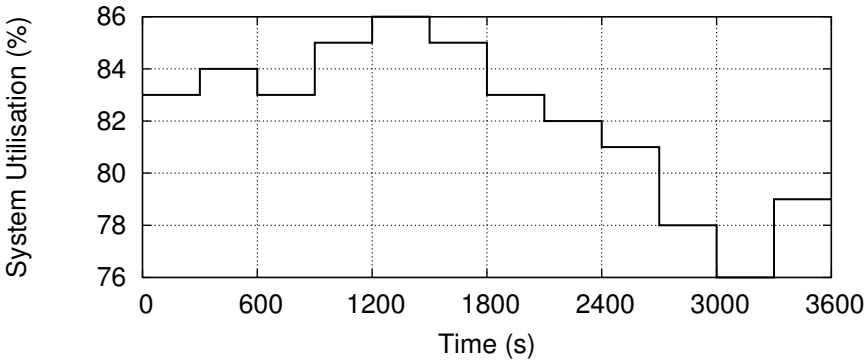


Figure 5.4: Total system utilisation.

In this paper, we use a predictive resource allocation algorithm (Algorithm 6, Chapter 4) that uses an exponential moving average to forecast the workload for the next interval and allocates resources accordingly. The algorithm ranks applications in accordance with their importance (this may be governed by SLAs or a business metric) and greedily allocates resources to applications in order of priority.

We selected five timings for rack failures. Each rack was then failed at each of the timings in a separate simulation. The times selected for the failures are 645, 1245,1845, 2445 and 3045 seconds. Each simulation run contains a single failure at one of the stated times in a single rack. The total simulation time was 3600 seconds.

5.5.3 Expected Results

Based on our proposed framework, we expect the following result from our experiments, which we will verify in the results section (Section 5.6):

- We generally expect the balanced resource allocator (Algorithm 7) to have less failed requests due to the fact that it creates allocation which minimises the capacity loss for each application across each rack.

- We anticipate that the random allocator may yield better performance in rare situations, where a rack is lightly leaded with respect to a single application, though this is expected to be offset by a higher application exposure, i.e, a high value of $CapacityLoss$, for an application on the same rack.

5.6 Results

In this section, we present the results of our experimentation. The results presented give (i) details of the overall system impact, (ii) analysis for each failure timing and (iii) discussion of the effect of failures on each application. Any

difference is between the balanced and the random allocator, where a positive difference represents better performance for the balanced allocator.

5.6.1 Overall Results

Allocator	Failure1	Failure2	Failure3	Failure4	Failure5
Random	3.20	2.83	2.42	1.87	1.60
Balanced	3.16	2.75	2.27	1.85	1.60
% Imp.	1.12	2.87	5.95	1.09	0.02

Table 5.1: Maximum total request failure percentage for each failure.

We now present the overall impact of a rack failure on the total failure rate over the duration of the experiment. Firstly, we consider the impact of the allocation technique on the maximum percentage of failed requests for each failure. Each value shown in Table 5.1 reflects the percentage of failed requests over the duration of a simulation. The results demonstrate that the balanced allocator reduced the maximum impact of a rack failure on the overall failure rate, as compared with the random allocator, as it consistently yields an improvements in the percentage of failed requests. The maximum observed improvement is nearly 6%. This is a considerable improvement, given that it was measured over the full duration of a simulation, which dilutes the number of failed requests across the total number of request during the simulated duration.

The standard deviation of the overall failure percentage for each rack and failure is shown in Table 5.2. The nature of the balanced allocator caused a consistent reduction in the standard deviation of the failure rates. This can be explained by the fact that the balanced resource allocator (Algorithm 7) attempts to reduce each application's capacity loss, thereby reducing the deviation in terms of failed requests. The most significant improvement can be seen in Failure 3, though it should be noted that the lowest percentage improvement is nearly 8%.

Allocator	Failure1	Failure2	Failure3	Failure4	Failure5
Random	0.0588	0.0373	0.0710	0.0433	0.0105
Balanced	0.0475	0.0309	0.0358	0.0237	0.0096
% Imp.	19.1851	17.1305	49.4922	45.2765	7.9847

Table 5.2: Standard deviation of failures across all racks.

5.6.2 Failure Results

Failure 1 in our experimentation occurs at 645 seconds, after two migration intervals. Table 5.3 contains the failures observed for each application, under both the random and balanced allocators. As mentioned before, we crashed each of the 10 racks in separate experimental runs at the 645 seconds mark. The minimum/maximum columns represent the minimum/maximum number of failed requests across all 10 failures. The rightmost column of Table 5.3 presents the percentage improvement in standard deviation of the balanced allocator over the random allocator.

We make the following two observations. For all applications, the minimum number of failed requests under the random allocator is equal to or better than the balanced allocator, and in terms of maximum values, the balanced allocator performs better than the random allocator. The first observation is due to the fact that the random allocator can place fewer instances of an application on a rack than the balanced allocator, i.e., an application can have a capacity of 0 on a rack, which never happens in the balanced algorithm (unless an application has no servers allocated), than the balanced allocator. On the other hand, for the maximum values, the random allocator may allocate servers in a rack to applications in such a way that the capacity loss is very high, resulting in very high number of failed requests.

Table 5.3 shows the difference in standard deviation between the two allocators. The result for Application 5 is due to a minor difference in the round robin scheduling of requests across servers. Application 1 has an identical failure rate under both allocators. The best and worst rack configurations for each application under both allocators are shown in Figure 5.5. The maximum variation

Table 5.3: Application results for Failure 1.

	Random				Balanced				
Application	Min	Max	Average	Std. Dev	Min	Max	Average	Std. Dev	Improvement (%)
1	1995	2494	2145	240.83	1995	2494	2145	240.83	0.00
2	2015	3526	2770.5	593.26	2518	3023	2770.5	265.53	55.24
3	2244	43213	19794	15027.35	12248	22572	17410	5440.17	63.80
4	794	1589	1350	277.62	1191	1589	1350	205.05	26.14
5	1034	1379	1275	166.08	1033	1379	1275	166.77	-0.42
6	1223	2446	1875	438.36	1630	2039	1875	210.65	51.95
7	507	14749	2083.5	4456.62	507	1015	811.5	262.07	94.12
8	531	21216	4987	8563.51	1062	1594	1275	274.12	96.80
9	78896	120352	95478.4	13990.33	89260	99624	95478.4	5351.95	61.75
10	0	20332	6293.6	8381.72	564	10008	4341.6	4876.86	41.82

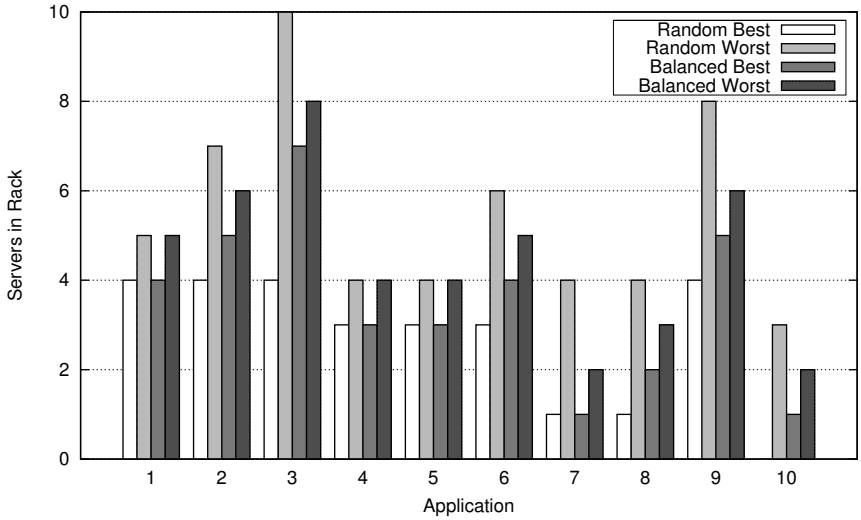


Figure 5.5: Range of allocations giving best and worst results for failure 1.

in allocations for the random policy is 6, while the balanced policy achieves a maximum variation of 1, which shows that Algorithm 7 of the balanced allocator achieves better "robustness" balancing than the random allocator.

This pattern of results is corroborated by the results from failures 2-5, which are given in Tables 5.4-5.7. This supports the first of our expected results, as proposed at the end of Section 5.5

5.6.3 Application Results

Table 5.8 gives the range of improvement provided by the balanced allocator across all failures and across all racks. The benefit of the balanced allocator varies between -0.42% and 96.80% across each application.

Table 5.4: Application results for failure 2.

	Random				Balanced				
Application	Min	Max	Average	Std. Dev	Min	Max	Average	Std. Dev	Improvement (%)
1	2086	2609	2347.5	275.22	2086	2609	2347.5	275.22	0.00
2	2012	4025	2817	679.31	2515	3018	2817	259.49	61.80
3	2026	5062	3900	894.12	3545	4053	3900	244.74	72.63
4	703	1407	1090.5	259.53	1055	1408	1090.5	111.56	57.02
5	904	1809	1537.5	316.39	1356	1809	1537.5	233.24	26.28
6	22848	64304	46685.2	10979.10	43576	53940	46685.2	5006.29	54.40
7	523	17533	2433.5	5311.79	523	1047	889.5	252.91	95.24
8	0	9060	1856	2590.29	1000	1500	1200	258.20	90.03
9	41822	83278	61513.4	11405.61	52186	62550	61513.4	3277.31	71.27
10	0	1046	732	441.01	522	1046	732	270.25	38.72

Table 5.5: Application results for failure 3.

	Random				Balanced				
Application	Min	Max	Average	Std. Dev	Min	Max	Average	Std. Dev	Improvement (%)
1	2095	3143	2514	331.34	2095	2619	2514	220.83	33.35
2	2044	4088	2862	689.52	2555	3067	2862	264.01	61.71
3	1784	4459	3300	703.31	2675	3568	3300	311.94	55.65
4	1073	5906	2361.5	1880.91	1073	5907	1985.5	1388.15	26.20
5	31948	52676	45421.2	8532.40	42312	52676	45421.2	5006.29	41.33
6	852	2556	1875	500.00	1704	2131	1875	219.90	56.02
7	494	17911	2433.5	5443.78	494	989	889.5	208.45	96.17
8	437	1751	1050	422.90	874	1313	1050	226.35	46.48
9	2064	3615	3045	513.64	2580	3098	3045	163.39	68.19
10	0	1326	619.5	427.38	442	885	619.5	228.51	46.53

Table 5.6: Application results for failure 4.

	Random				Balanced				
Application	Min	Max	Average	Std. Dev	Min	Max	Average	Std. Dev	Improvement (%)
1	2064	3611	2631	451.50	2579	3095	2631	163.03	63.89
2	2037	4075	2904	681.62	2547	3058	2904	246.12	63.89
3	1688	4226	3000	703.48	2535	3381	3000	240.17	65.86
4	843	2110	1350	387.89	1265	1688	1350	177.88	54.14
5	32968	64060	51623.2	8175.23	43332	53696	51623.2	4369.85	46.55
6	11282	52721	36145.6	12160.21	32001	42362	36145.6	5349.79	56.01
7	427	2135	811.5	549.43	427	855	811.5	135.10	75.41
8	443	1773	975	407.36	886	1329	975	186.57	54.20
9	1308	3055	2355	512.30	2180	2617	2355	225.28	56.03
10	0	3567	778.5	1028.41	468	938	562.5	197.64	80.78

Table 5.7: Application results for failure 5.

	Random				Balanced				
Application	Min	Max	Average	Std. Dev	Min	Max	Average	Std. Dev	Improvement (%)
1	2032	3556	2692.5	481.90	2540	3048	2692.5	245.32	49.09
2	2030	4063	2944.5	710.41	2537	3047	2944.5	214.51	69.80
3	1860	4647	3300	772.55	2788	3718	3300	263.86	65.85
4	681	1704	1090.5	313.16	1022	1364	1090.5	144.15	53.97
5	1964	2455	2062.5	206.87	1964	2456	2062.5	207.39	-0.25
6	852	2557	1875	500.19	1704	2131	1875	220.12	55.99
7	357	1784	678	458.89	357	714	678	112.79	75.42
8	500	1500	1050	437.80	1000	1500	1050	158.11	63.88
9	1168	2337	1869	402.15	1557	1947	1869	164.44	59.11
10	0	2553	1075.5	1031.49	516	2554	923.5	858.82	16.74

Table 5.8: Application results across all failures and racks (percentage change in standard deviation).

App.	Min (%)	Max(%)	Average(%)
1	0%	63.89%	29.27
2	55.24	69.80	62.49
3	55.65	72.63	64.76
4	26.14	57.02	43.49
5	-0.42	46.55	22.70
6	51.95	56.02	54.87
7	75.41	96.17	87.27
8	46.48	96.80	70.28
9	56.03	71.27	63.27
10	16.74	80.78	44.92

In two cases Application 1 gains no benefit form the allocator, due to identical allocations from both policies. The balanced allocator improves the deviation of Application 1 by an average of 29.27%.

When using the balanced allocator Application 5 has a marginally worse variance (-0.42% and -0.25%) than the random allocator. This is due to the round-robin scheduling of requests to servers causing a slightly higher load at the point of failure.

5.7 Discussion

The results presented have demonstrated the need for failure awareness to be incorporated into systems which operate under DRA policies. The most desirable approach for achieving this would be one that requires no modification to the DRA policies already in use, i.e., the most desirable approach would be composing the DRA policy itself with a failure-aware allocator component. The modular architecture proposed in this paper has shown that, by decoupling the allocation mechanism from the DRA policy, the performance-oriented goals of the DRA can be separated from the conflicting aim of improving application robustness.

Once the decoupling of the DRA policy and allocation mechanism has been achieved, the focus turns to the function of the allocation mechanism itself. The

metric proposed in this chapter, *capacityLoss*, serves to quantify the exposure of an application, e.g., where $capacityLoss = 1$ for a given application, that application is completely exposed to a failure of the rack on which it is hosted. This focus on application exposure is motivated by the fact that, while it is reasonable to balance applications across racks in the context of static allocation, this is not possible under existing DRA policies, which assume the existence of a common, un-partitioned resource pool.

The proposed modular architecture and balanced allocation mechanism have been shown to reduce the impact of failures on applications being serviced across racks. Interestingly, the random allocation mechanism was shown to provide the lowest absolute impact for any single application when the failed rack was lightly loaded with respect to the application, i.e., when the capacity loss of that application on a rack was minimal. However, this is not necessarily positive, as for this light loading to occur, another rack may be heavily loaded. Results averaged across all racks demonstrate significant improvements in the mean number of failed requests for the balanced allocator. In addition to the average case, the balanced allocator exhibited lower deviations than the random allocator. Since rack failures are inherently unpredictable, minimising the average case is clearly of great benefit when operating at large-scale. In turn, this represents lesser exposure to loss of income due to unforeseen unavailability of applications.

As the magnitude of the systems operating under DRA policies increases, the scale and frequency of the problems addressed in this paper will similarly increase. Hence, as failures become the expectation rather than the exception for large-scale systems, effective resource allocation and modular architectures that facilitate a separation of concerns will become increasingly important for cloud providers offering Infrastructure as a Service.

5.8 Summary

In this chapter, we have made the following novel contributions: (i) we have presented a modular architecture for failure-aware resource allocation, where a performance-oriented DRA policy is composed with a failure-aware resource allocator; (ii) we have proposed a metric, called *CapacityLoss*, to capture the exposure of an application to a rack failure; (iii) developed an algorithm for reducing the proposed metric across all applications in a system, and (iv) evaluated the effectiveness of the proposed architecture on a large-scale DRA policy in context of rack failures in order to show the efficiency of our approach.

The results presented demonstrate the effectiveness of our architecture and mechanisms, with consistent improvements being observed in almost all situations. Indeed, when the average case is considered, the proposed approach exhibits a minimum improvement of more than 22% and a maximum of more than 87% across all failure situations.

The novelty of our modular, failure-aware architecture for resource allocation is that it is applicable to, and will work in tandem with, any DRA policy. Practically, this implies that organisations need not modify their DRA policy, rather they can just integrate the failure-aware resource allocator to reduce the exposure of applications to rack failures.

CHAPTER 6

A Model for Resource Allocation Cost

The reallocation of resources between tasks is not instantaneous and as a result the resource being reallocated is unavailable to do useful work for the period of reallocation. This loss of capacity of the overall system may be deemed a cost to the provider of the platform.

Chapter 4 presented a heuristic which offered improved scaling characteristics over other resource allocation algorithms. However the heuristic presented was naïve, as it did not take into account the potential reallocation cost for its migration. The cost of the migration is not fixed, and grows depending on the total number of migrations across the whole system. This chapter will develop a generic model for resource reallocation that is suitable for use with many dynamic provisioning policies. In this chapter the cost of a migration is considered to be the time taken for the new allocation to occur, as this metric may be easily translated into other metrics as desired (i.e. unserviced requests).

6.1 Specific Related Work

The automated installation and configuration of resources is a valuable tool for system administrators. It provides a consistent and dependable mechanism for the deployment of new hardware. Network installation mechanisms for operating systems, e.g. Jumpstart for Solaris [5] and the fully automated installer [3] for Linux, have existed for a long period of time. These mechanisms can install an operating system over the network with little human interaction. These tools work well in situations where machines require an identical image but become cumbersome where many different configurations or variations exist.

An example of an early infrastructure wide management tool would be Cfengine [23] initially developed in 1993 which offers a generic platform for the central management of dedicated infrastructure, including network interface configuration, mounting network filesystems and modification to local system files.

In cloud platforms such as Nimbus [70], Open Nebula [54] and Eucalyptus [36] virtual machine images containing a full software stack of operating system, libraries and applications is deployed. Work in [61] offers a comparison of the three cloud platforms and describes the process for VM deployment in each.

There has been much research into the performance of allocation of resources [17, 75]. In [17] the authors develop a benchmarking process for the deployment of VM images in both Xen [6] and KVM [4] paravirtualised I/O environments. In this chapter a generic model of the process is developed, for use by DRA policies. In [75] the authors examine the impacts of system utilisation on the source and target machines of a live VM migration. It is demonstrated that a source machine with high CPU utilisation has a significant effect on the migration duration. The model presented here is based on a cold deployment of a VM rather than a migration of a server, in order to present the addition of a new VM.

Additionally, the effects of hardware caches underlying virtual machines is discussed as a note in [74]. In this work hardware cache flushes have a negative impact on the throughput of an application during a live migration.

[77] explores the impact of a live migration of a VM between two physical hosts. The approach taken here differs, as it considers the reallocation of an existing slot on a physical host, which is to say that this work considers the undeployment time in addition to the deployment.

6.2 System Model

In this chapter two different migration mechanisms are adopted which have both been observed in the literature. In the first a dedicated deployment is considered in which a server is fully assigned to an application, and a reinstallation of the entire software stack may be required. The second mechanism which is explored is the deployment of a virtual machine which is assigned to a physical host running a hypervisor. This environment is common in IaaS infrastructures. These mechanisms are explained in more detail in Sections 6.2.1 and 6.2.2.

6.2.1 Dedicated Deployment

Under a dedicated deployment model, a resource migration may require a server to be completely re-imaged. In our test cluster, the Debian based Fully Automated installer is used to manage system images. The process for re-imaging a server under this scheme is:

1. Write FAI configuration file containing details of software packages and post-install scripts required to build a system correctly.
2. Enable the configuration file for the host to be imaged.
3. Reboot the host.
4. The host requests an IP address via DHCP.

5. After receiving an IP address, the host queries the TFTP server for a kernel image which is then booted.
6. The minimal kernel queried the FAI install server for the correct configuration and details of the package server to use.
7. Local hard disks are partitioned.
8. Software is installed via the standard apt-get mechanisms.
9. Post install scripts and OS configuration.
10. Log files are saved to install server and the system is rebooted.

In such environments it is common to have a local package mirror to reduce external bandwidth requirements and also increase the installation speed.

6.2.2 Virtual Machine Deployment

A virtual machine image typically contains a complete software stack, from OS and libraries to applications. Thus a cloud deployment can be modelled by the generic model by considering a single stage for deployment and undeployment. To explore the costs for a virtual machine deployment, Nimbus was selected as the cloud platform.

The nimbus architecture can be seen in Figure 6.1. Clients request new virtual machines from the service node which maintains a repository of virtual machine images. These images are pushed to the physical hosts which comprise of a host OS with a virtualisation enabled kernel, a small nimbus service, libvirt to enable the programatic control of any host VMs and Xen as a virtualisation platform. Once the image is received by the physical host it is booted by the nimbus service through libvirt. Once booted the virtual machine requests an IP address which makes it available to users over the network. Access control and resource limitations for users are configurable on the service node.

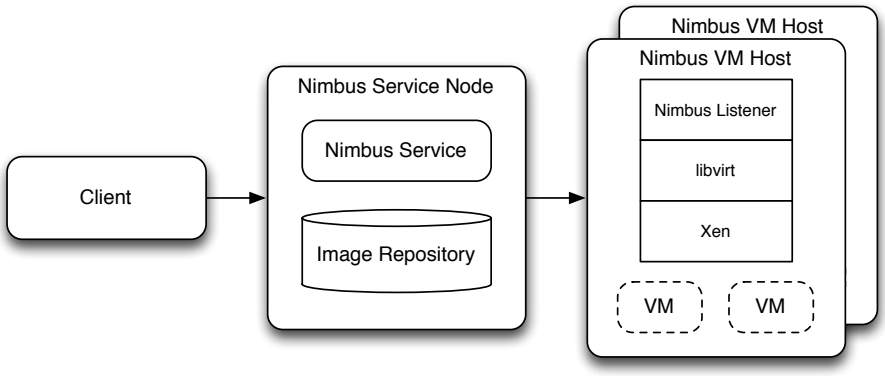


Figure 6.1: An overview of the Nimbus architecture.

Under this deployment mechanism, it is clear that a significant duration may be spent in the network transfer of the full virtual machine image, before the virtual machine may be used.

6.3 The Cost Model

The cost model is designed to be generic, such that components may be omitted where they are not required. The stages of the model are derived from the switching process as described in Chapter 3 and the process outlined in [13]. An overview of the cost model is shown in Figure 6.2.

The cost model is composed of a multistage process. The key stages are:

- Completion of queued requests. This allows requests currently being processed and those currently queued on the server to be processed. This avoids the need to drop requests.
- Undeployment of an application. This stage of the model accounts for the process of removing an application or a virtual machine from a server.
- Deployment of a new application. This stage of the model captures all facets of installing a complete application inclusive of supporting programs and libraries.

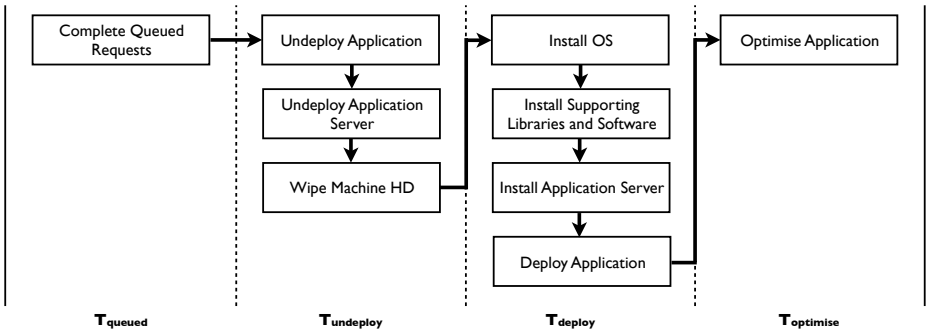


Figure 6.2: Cost Model Process Overview

Parameter	Description	Units
X	Throughput of service	Requests per second
R	All requests in a system	
r	A single request type	
n	Number of requests in system	
p_r	Proportion of requests of type r	Percentage
S	All components in a software stack	
s	An individual software component	
h_s	The time taken to terminate component s	Seconds
w	The time taken to erase a hard disk	Seconds
t_{init}	Time taken to initialise reconfiguration	Seconds
$t_{install}$	The time to restart a server for network install	Seconds
t_{boot}	The time to restart a server after installation	Seconds
t_s	The time to install software s	Seconds
c	The number of concurrent deployments	
d_s	The total data transfer of a software component.	Megabytes
b	The minimum available network bandwidth	MB per second

Table 6.1: Model parameters.

- Optimisation of an application. After deployment an application may require a period of "warm up" time to deliver similar performance to its peers.

6.4 Calculating a Stage Cost

In this section the method for calculating the cost of each stage is given. The notation used in this chapter is shown in Table 6.1.

6.4.1 Calculating T_{queued}

An approximation of the duration for the completion of queued requests may be calculated using equation 6.1. This works well when the service durations for all requests types are similar (e.g. a Web application), or only limited system information is present. Where more detailed information about the distribution of requests is available equation 6.2 would provide a more accurate prediction.

$$T_{queued} = \frac{n}{X} \tag{6.1}$$

$$T_{queued} = \sum_{r \in R} \frac{np_r}{X_r} \tag{6.2}$$

6.4.2 Calculating $T_{undeploy}$

This component captures the time taken to stop and remove an application from a server. Applications may simply need to be stopped and undeployed from within an application server (as in chapter 3) or may require all components of the software stack to be stopped gracefully and removed. In the model we assume that applications do not save state, which is a reasonable assumption in the case of software designed for horizontal scaling. Equation 6.3 details the calculation involved in the undeployment of an application.

$$T_{undeploy} = \sum_{s \in S} (h_s) + w \tag{6.3}$$

6.4.3 Calculating T_{deploy}

Where servers are installed from bare metal, the duration of the install process is dominated by the time taken to configure the system and write the new software stack to the disk. In this chapter we consider the process of re-imaging a machine to require a reboot to initiate the process, and a subsequent reboot once the imaging of the machine is completed.

$$\text{(Dedicated) } T_{deploy} = t_{init} + t_{install} + \frac{c\sum_{s\in S} d_s}{b} + \sum_{s\in S} t_s + t_{boot} \tag{6.4}$$

Where the deployment consists of preconfigured VM images, the significant component of the deployment time is the time taken to transfer the image from a repository to the target, and then start the VM. In this case there is a single software component, s and t_{boot} is taken as the VM startup duration. Equation 6.4 may then be simplified to equation 6.5.

$$\text{(Virtual) } T_{deploy} = t_{init} + \frac{cd_s}{b} + t_{boot} \tag{6.5}$$

6.4.4 Calculating $T_{optimise}$

After deployment an application may take a period of time to deliver performance equivalent to its peers. This stage is clearly application dependant, so no equation is given here.

6.5 Experimental Setup

In this chapter both forms of the model are examined. In this section the experimental setup for the work is described.

6.5.1 Dedicated Deployment Setup

In the case of a dedicated machine deployment, two different software stacks are used. The details of the hardware platforms can be seen in Table 6.2. Each of the software stacks are installed using FAI[3] with additional post-install scripts where required.

- **HPC Software Stack** This stack is comprised of a stripped down Debian[2] linux installation and software libraries (e.g. MPI) to support parallel scientific computation.

	Physical Deployment	Cloud Deployment
Processor	Intel Xeon 2.6 GHz	Intel Core2 quad core 2.4 GHz
Processor Count	2	1
RAM	2GB	4GB
Storage Subsystem	IDE	SATA
Storage (GB)	80GB	80GB
Network	100Mbps Ethernet	1Gbps Ethernet

Table 6.2: Benchmark hardware platforms.

- **Web Application Stack** This stack uses the same linux installation as the HPC stack, but adds a java runtime, Glassfish application server and JDBC libraries to support a multitude of database platforms.

6.5.2 VM Deployment Setup

To examine VM deployments in a cloud scenario Nimbus is deployed as the cloud management tool. The management server is a 16 core 2Ghz AMD Opteron with 32GB of RAM and 1TB of storage (configured as a RAID device, mirrored across two disks). It has a single 1Gbps network port dedicated to image distribution. The specification of the eight host nodes is given in table 6.2. The nodes are connected to the management server by a 1Gbps switch dedicated to the test platform.

6.6 Model Parameters

To measure all parameters in a controlled way, a script was developed to perform migrations and collect statistics. This allowed for repeatable experiments. The process of enabling a reconfiguration through the FAI mechanism (t_{init}) was found to be 0.076 seconds and was measured through timing components in the script. The boot times (t_{boot}) for the machines were measured by performing a reboot remotely, and measuring the time taken until the machine was available remotely via SSH.

For the physical deployment the size of the data transferred during a migra-

Parameter	HPC Software	Web Application
X	-	165 requests/s
t_{init}	0.076 seconds	
$t_{install}$	26.5 seconds	
t_{boot}	57.1 seconds	88 seconds
$\sum_{s \in S} s_d$	408.34 MB	210.7 MB
b	11.1 MB/s	
$\sum_{s \in S} t_s$	-	115.97 seconds

Table 6.3: Physical deployment parameters.

Parameter	Value
t_{init}	8 seconds
t_{boot}	5.7 seconds
s_d	500MB
b	108 MB/s

Table 6.4: Cloud deployment parameters

tion was measured using the statistics on the network switch. This was done to capture traffic across all network protocols used in the FAI process. The bandwidth measurement was obtained by measuring the time taken to send 50 MB across the network from the install server to the client machine.

The additional startup time for the Web application is primarily due to the time taken to start the Java application server and deploy the application.

The initialisation time for the cloud deployment was found by measuring the time between requesting a new virtual machine deployment and the server beginning the transmission of the image across the network. The boot duration for a virtual machine was measured by starting an virtual machine image on a host directly. Bandwidth in the cloud network was measured by timing the transfer of the 500MB image from the image repository to the host server via SCP, which is the mechanism used by Nimbus.

All parameters were measured several times and averaged to ensure irregular values did not skew the model. The physical and cloud deployment parameters are shown in Table 6.3 and 6.4 respectively.

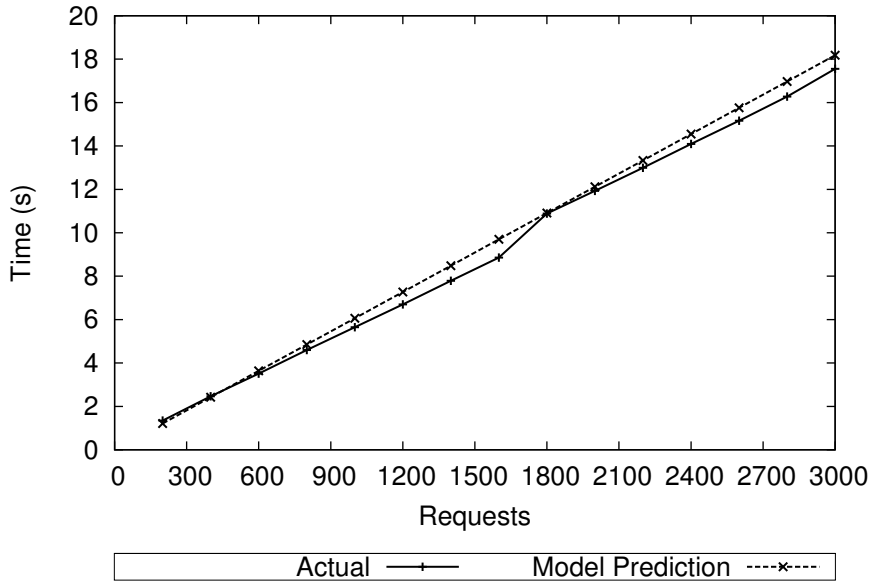


Figure 6.3: T_{queues} for Web application.

6.7 Results

In the case of a HPC server resource being migrated, the worst case time for completion on current tasks would be the wall time limit of the task as it is set by the user or scheduler. In the case where multiple jobs are running on a single machine this would be the maximum wall time of all jobs. The Web application may have a number of requests in flight or queued. The graph in Figure 6.3 gives an insight into the accuracy of the model for this phase of the migration process.

In each of these environments, the undeployment duration is ignored as there is no requirement for a graceful termination of the processes.

The results for the deployment phase of the model are given in Figure 6.4. The model performs well for the physical deployment, with errors less than 10.5% in all cases (see Table 6.5). The model delivers better accuracy for the HPC deployment than the Web Application deployment.

The optimisation of an executing application will vary depending upon the application in use. In our test environment the time taken to optimise the

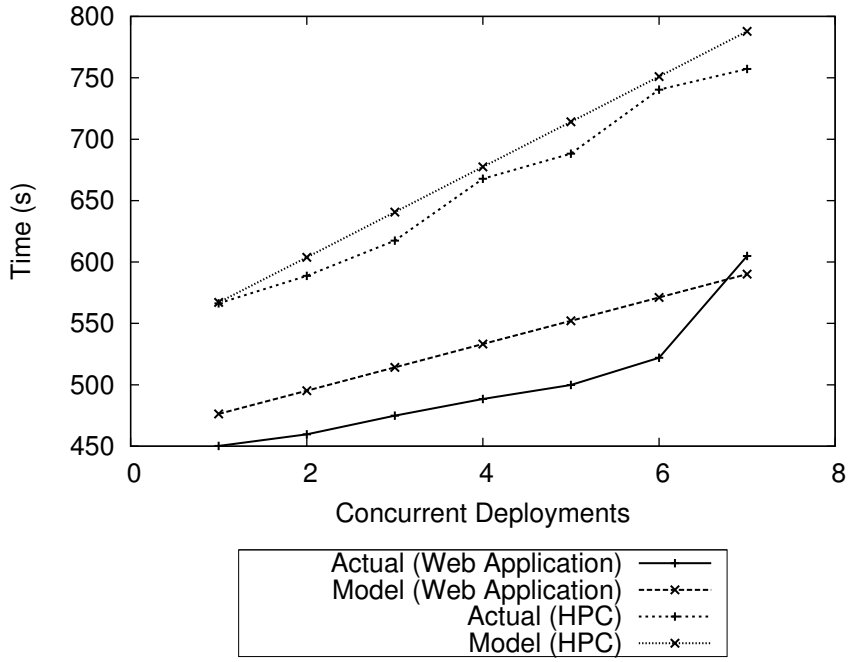


Figure 6.4: T_{deploy} for each environment.

Concurrent Deployments	Web Applications			HPC Software		
	Actual	Model	Error (%)	Actual	Model	Error (%)
1	450.27	476.24	5.77	566.29	567.07	0.19
2	459.69	495.22	7.73	588.77	603.85	2.56
3	474.90	514.20	8.28	617.45	640.64	3.76
4	488.44	533.18	9.16	667.83	677.43	1.44
5	499.86	552.16	10.46	688.24	714.22	3.77
6	522.06	571.15	9.40	740.34	751.00	1.44
7	604.85	590.13	-2.43	757.21	787.79	4.04

Table 6.5: Error in physical deployment model.

system in the case of the Web stack running the sample Daytrader application is a single request. Testing shows an initial request to the system takes between 3077 and 4418ms, while the subsequent requests are served in less than 50ms. This initial startup time is due to the application server acquiring database connections in the first instance, which it holds once connected. Additionally after the first request static assets (e.g. images and stylesheets) are cached in memory, and as a result are not required to be fetched from disk.

The model for the physical deployment consistently over estimates the actual time taken for a deployment. Currently we believe that this is due to the way in which software packages are installed on the machine. Under this installation mechanism, a package is requested and then installed which leads to an interleaving of requests and installations across concurrent deployments. This reduces the network contention, which makes real deployments faster than the model predicts.

The graph in Figure 6.5 demonstrates the accuracy of the model for the deployment phase in a cloud environment. The linear nature of the results is due to the limitation of the network bandwidth. Table 6.6 presents the error in the model. In all cases the model is accurate to more than 9%.

The model consistently underestimates the duration for a migration. It is believed that the underestimation is due to network contention between the concurrent deployments.

6.7.1 Virtual Machine Model

The cloud deployment delivers a total duration for all migrations. However there may be significant variation in the time to to deploy individual VMs. Figure 6.6 demonstrates the average time taken to transfer images via SCP alongside the upper and lower bounds for the transfer. It is clear that there is a large variation in the time taken to transfer individual images, some VM transfers completing an order of magnitude faster than the average, and maximum, transfer time.

There are a number of deployments where the minimum observed transfer

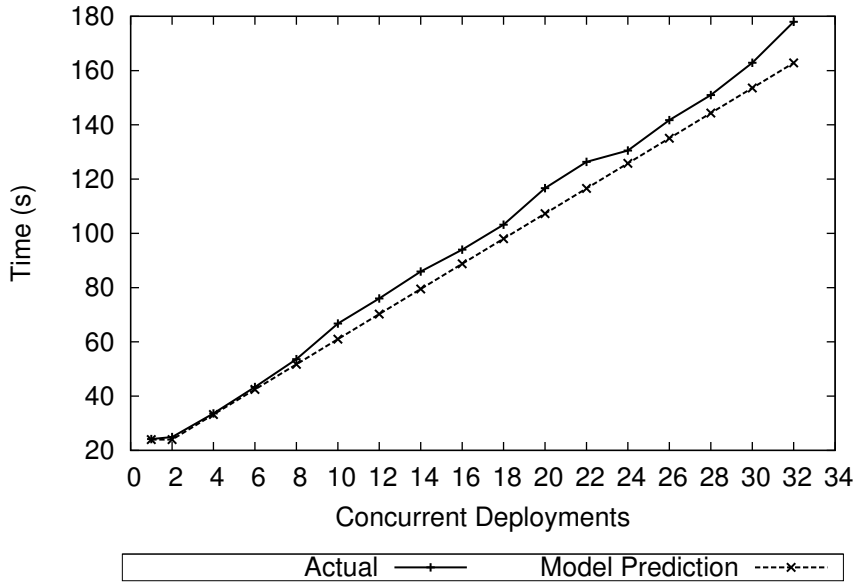


Figure 6.5: Cloud deployment model results.

Concurrent Deployments	Actual (seconds)	Model (seconds)	Error (%)
1	24.16	23.96	-0.84
2	24.92	23.96	-3.86
4	33.63	33.22	-1.21
6	43.37	42.48	-2.06
8	53.63	51.74	-3.53
10	66.75	61.00	-8.62
12	76.00	70.26	-7.56
14	85.95	79.51	-7.48
16	94.01	88.77	-5.57
18	103.22	98.03	-5.02
20	116.65	107.29	-8.02
22	126.37	116.55	-7.77
24	130.53	125.81	-3.62
26	141.75	135.07	-4.71
28	151.00	144.33	-4.42
30	162.88	153.59	-5.70
32	178.00	162.85	-8.51

Table 6.6: Error in cloud deployment model.

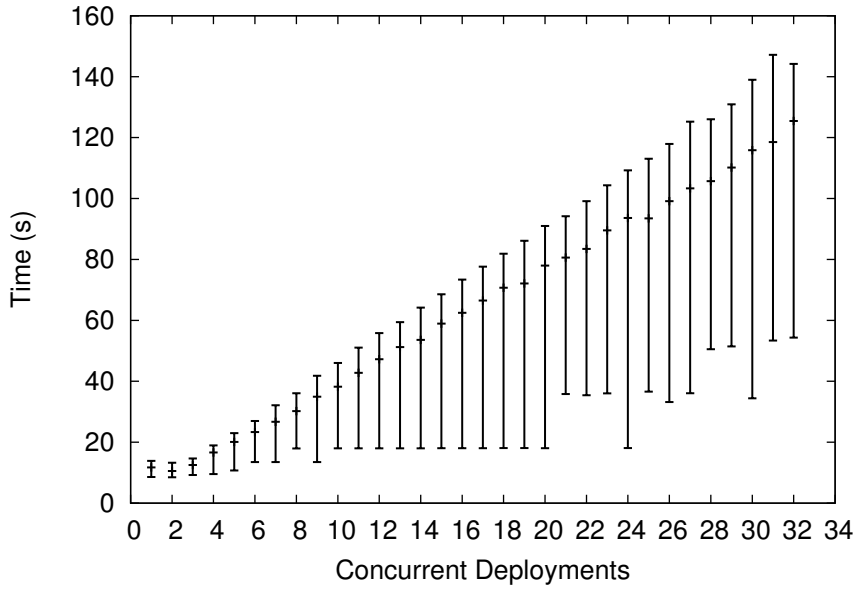


Figure 6.6: Virtual machine transfer variation.

time is around 18 seconds. This occurs at concurrent deployments of 10-20 and 24 virtual machines. In other cases the minimum transfer duration is significantly lower than the average and maximum duration. It is believed that this is an effect of the broker application processing deployment requests in an imbalanced way between all deployments.

6.8 Impact of the Cost Model

With such a large variation in migration duration, it is clear that migration cost can no longer be considered fixed. To asses the impact of a realistic cost on the behaviour of a real policy. The Average Flow policy considers the cost of switching servers between pools as a reduction on the number of requests which can be served by the system in total. However, as the number of simultaneous deployments increases, so does the length of time taken to do the migration. This increase in migration time affects the capacity of the system for a longer period, thus increasing the cost of each migration. In the first instance we analyse the Average Flow policy (see Algorithm 2) under the parameters listed

	Application 1	Application 2
Completion Rate (per server)	100	
Job Cost	50	50
Queue Length	20	20
Server Allocation	50	50
Switches in Progress	0	0

Table 6.7: Experimental Parameters.

in Table 6.7. Parameters for both applications are kept the same and only workload and is varied.

A total workload of 7500 requests per second across 100 servers is used which gives an overall utilisation of 75%. We then divide the workload between the applications in intervals of 10% and asses the number of migrations made by the policy using both a fixed cost and the cost model which we have defined above.

The fixed cost model uses a cost fixed at the cost of a single migration, while the dynamic cost model works using the VM migration scheme above with a VM size of 500MB. The hardware platform used for the prediction is that benchmarked for the VM cost model.

The results of this experiment are shown in Figure 6.7. The number of migrations are symmetric as the parameters for each application are the same. At the more extreme workload distributions it is clear that the increased cost of each migration outweighs the perceived benefit of the migration. This results in fewer migrations being made by the policy when using the cost model.

The difference between the cost model varies depending on the total system utilisation. At higher system utilisations, migrating servers becomes more necessary. Figure 6.8 demonstrates the largest difference between the cost models are utilisations between 60% and 95%. Below 60% utilisation no migrations are made by the policy.

The graph demonstrates that at utilisations up to 85% the cost model reduces the number of migrations within the system due to their increased total cost to the system. However beyond 85% utilisation the Average Flow effects

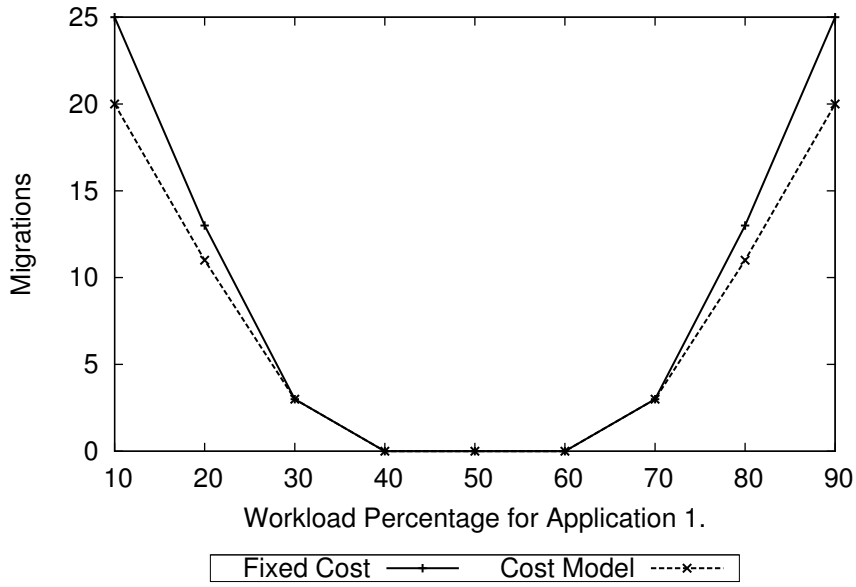


Figure 6.7: Impact of the cost model on the Average Flow policy.

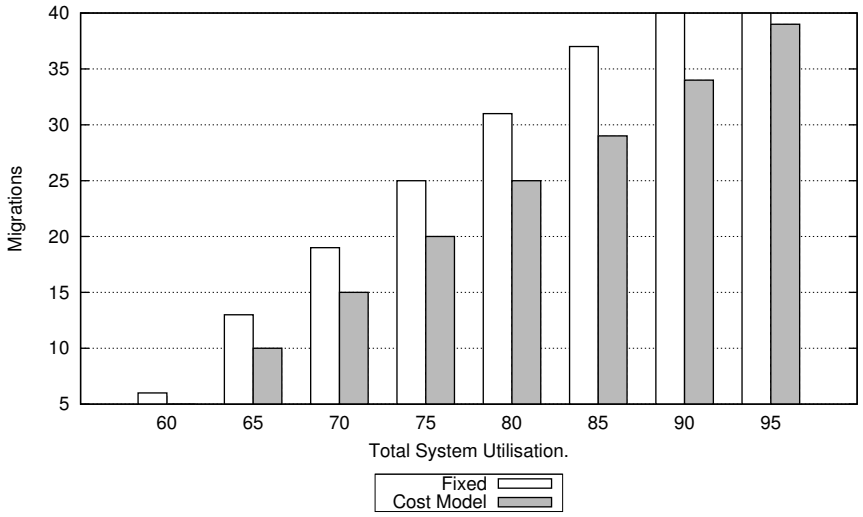


Figure 6.8: Maximum difference in migrations at high utilisation.

similar migrations under both cost models, as workload demands are so high when compared to the cost of the migrations.

6.9 Implications and Discussion

The model displays good accuracy for all phases of the model in both environments. The linear nature of the model in the deployment scenarios here is a feature of the scenarios themselves. Where high contention of VMs on a physical machine exists, the bandwidth to disk may become a limitation rather than the network capacity to a machine

While individual images can be compressed to reduce their size, mass migrations of virtual machines can incur a significant cost. The significant variation in individual deployment times was unexpected. This result may undermine the assigned prioritisation of applications in the algorithm presented in Chapter 4.

The limiting factor for both deployment scenarios is the network capacity, as it is the major point of contention for the resources being reallocated. This is an issue for centralised resource management systems. Nimbus includes support for a distributed hosting of images, however this is not the default configuration and it is not considered here.

The behaviour of the Average Flow policy is clearly affected by the cost of a migration. This indicates that an accurate cost is important when making migration decisions, so that resources are not over or under provisioned due to incorrect parameterisation of the resource allocation algorithm.

6.10 Summary

In this chapter we have explored typical deployment mechanisms for both dedicated and cloud infrastructure as defined in the literature and in practice. A model of resource reallocation cost has been developed, and shown to be effective to within 11% on dedicated platforms and 9% on virtualised platforms. The

model will aid in providing accurate details for resource provisioning algorithms.

Additionally we have used the cost model in conjunction with the Average Flow policy and demonstrated that the additional overheads of multiple concurrent deployments have an impact on the behaviour of the policy, which is not considered by the authors of the policy.

CHAPTER 7

Discussion and Conclusion

As the scale and scope of online services has grown so too has the infrastructure needed to support them. Outsourcing this infrastructure to commercial hosting providers has enabled developers of online services to benefit from specialised knowledge, experience and above all cost-effective deployments. The hosting provider must deliver an acceptable level of service in return for their fee. Dynamic provisioning has been shown to be effective in the management of resources. However much of the analysis to date has focussed on the performance delivered to the applications themselves exclusive of resource failures, an issue which cannot be ignored in large infrastructures, and migration costs.

In this chapter we offer some perspective on the research and present an architecture in which all of the contributions are included. Further we discuss avenues for future work.

7.1 Perspectives on the Research

There are a number of issues which the research in this thesis addresses. In this section we examine the contributions in light of recent developments and offer an insight into the implications of these developments.

7.1.1 Virtualisation

Much of the early work in this thesis is developed around a hosting provider that provides dedicated resources to clients. In recent years, the increased support for virtualisation in hardware and software has enabled the cost-effective provisioning of infrastructure as a service. If we now review our results in the context of virtualised servers, Chapter 4 offers a framework for scalable resource allocation, and as a result demonstrates readiness for the increased application density offered by a virtualised environment. The work in Chapter 5 discusses a failure aware allocator. In the case of a virtualised environment, the $CapacityLoss$ metric defined still applies in its current form as the virtual servers will be distributed evenly across all racks in the environment. The cost model has been evaluated in a virtualised environment and demonstrated suitable accuracy for use.

7.1.2 Scaling Beyond A Single Data Centre

The results in this thesis are primarily concerned with resource management by a hosting provider and have assumed all resources are contained within a single data centre. It is feasible that a large hosting provider may have multiple data centres in order to provide business continuity or disaster recovery services. The network bandwidth available between sites will be shared between resources at each site, which potentially creates a bottleneck. The network latency is also an issue which will increase as the distance between the sites increases.

Given that the failure-aware allocator takes a rack-level view of the infrastructure it is possible that servers may be allocated across all data centres, in

accordance with the *CapacityLoss* metric. This would provide improved robustness for the application at the expense of performance. The effect of cross data centre provisioning may be undesirable in applications which cannot tolerate imbalances in performance between replicas. Conversely work in [22] suggests that distributing machines near to their clients may have a positive effect on performance.

The cost model (Chapter 6) uses network bandwidth as a parameter in order to predict the duration of a server reallocation. Although untested in such an environment, it is expected to perform to a reasonable level of accuracy, as the model could be parameterised in accordance with the environment. We would expect the proportion of intra and inter data centre migrations to increase the variation in the individual imaging time of the virtual servers, and affect the maximum reallocation time accordingly. If each site maintains its own repository of virtual machine images then there would be no effect on the cost of migration as it would be done within a single data centre however, ensuring that all repositories are up to date would become an additional issue.

7.1.3 Middleware Platform

It may be desirable for a customer to use resources from multiple hosting providers in order to mitigate risk or prevent platform lock in. While this thesis has focussed on the role of the hosting provider, there is considerable provision for the services to be thought about from a customer's viewpoint. The ideas presented here could be used as a middleware platform for a customer to support multiple providers. The extent to which the ideas could be supported would are dependant on the support offered by vendors, e.g., an ability to identify the rack id for placement of a server.

Where multiple providers are used it may be possible to balance the characteristics of various providers, e.g., reliability and deployment time, to deliver an optimal solution for a given situation. As an example, servers which are less reliable but quicker to provision may be used in the event of a sudden workload

increase, with more reliable servers replacing these instances over time.

7.1.4 A Scalable Failure-Aware Architecture for the Cloud

Each of the contributions within this thesis has addressed an issue from Chapter 2. These may be assembled together to form an overall architecture which is more effective than those currently available. The overall proposed architecture can be shown in Figure 7.1. When compared to the platform assembled in Chapter 2 the combined architecture is much more mature. The key features of the combined architecture are improved robustness through the use of the failure aware allocator, improved scalability through the use of highly scalable resource allocation algorithms and aware of the total reallocation time via the cost model.

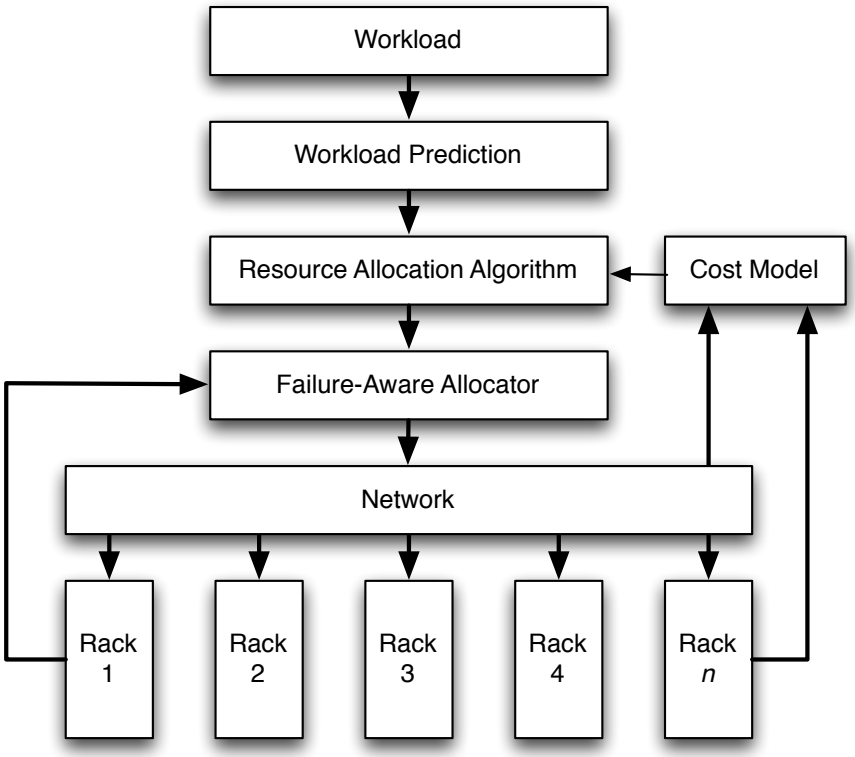


Figure 7.1: Overall Architecture

7.2 Research Contributions

Throughout this thesis dynamic resource allocation has been applied to enterprise applications. The algorithms have been demonstrated to work int this environment at small scale, however their inherent scalability issues limit their use in a modern data centre. This thesis has focussed on addressing issues with dynamic resource allocation at large scale.

To support this, the following specific contributions were made when developing a large-scale resource allocation architecture;

- A demonstration of a theoretically derived policy on a representative three tier application. The benefits of the policy were observed on two workloads, with a minimal overhead introduced by the switching platform.
- The exponential complexity of the state-of-the-art policies is explored and a formalisation of the NP-hard allocation is provided. A heuristic framework is developed with much improved scalability and an instance of the framework is tested against an established policy. The framework demonstrates significant improvements in the successful completion rate for managed applications.
- A robustness metric is devised to reduce the exposure of applications to rack-level failures. The metric is developed into a resource allocation component which can be used to augment any resource allocation algorithm, with minimal overhead over a random allocator.
- A modular model for resource reallocation cost is developed. This model can be used to predict the increased cost of concurrent resource migrations, which is an important consideration for many resource allocation algorithms. The cost of resource migration is shown to have a significant impact on the behaviour of resource allocation algoirhtms.

The contributions above represent novel work in the field of dynamic resource allocation for enterprise systems, and can be drawn together to develop

a scalable failure-aware resource allocation architecture.

7.3 Thesis Limitations

Dynamic resource allocation may be considered from a variety of perspectives. Primarily this thesis is concerned with issues that affect the use of dynamic provisioning at scale. The analysis of these issues is conducted mainly in the performance and request failure implications of both the resource provisioning mechanism and the applications supported by the dynamic infrastructure.

The resources which are managed by the dynamic resource allocation system are considered to be homogeneous. This is a reasonable assumption where new platforms are purchased at the same time, however over time systems may need to be replaced which are not of equal performance. Considering newer resources to be of equal performance as older resources may lead to their under utilisation as newer systems are invariably more performant than those which they replace, due to the rate at which technology progresses.

Applications in this work have been considered to be stateless. The deployment and migration of applications has not required state to be maintained. While this is certainly the case for lightweight web applications, database servers with large quantities of data may incur significant migration overheads.

Additionally applications executing on the dynamic platform are all considered to be independent. This consideration is reasonable where the platform is managed by a hosting provider, however the deployment of dependant applications or services to a cloud poses additional questions about platform management and service continuity in the event of resource failures. This would form another angle to extend the work presented here.

7.4 Further Work

The application of dynamic resource allocation to enterprise systems is not new. However, a novel aspect of this work is the consideration of resource failures within a large infrastructure. In the current work we have developed a failure-aware allocator for mitigating the effects of rack-level resource failure. As a next step we are assessing the suitability of spare capacity in the system as a mechanism for handling faults. The hope is that where space capacity exists, global reallocation can be avoided. In conjunction with this the potential for partial reallocation within an infrastructure, e.g., a network subtree is also being explored.

An issue not considered here are resource failures during a period of re-allocation. The issues of a large resource failure during migration are many, but the primary issue is that of inconsistent states between the expected state after reallocation and the state of the system due to the failure experienced. Our future work will look to address this inconsistency by re-evaluating the resource allocation and adapting it to suit the current environment with minimal overhead.

The power consumption of large data centres is a current hot topic. Dynamic resource allocation has the potential to reduce overall power consumption by consolidating applications of virtual servers onto fewer physical machines when the load allows. The reduction in power consumption has a clear benefit to the hosting provider. Power consumption will be explored in future work.

Bibliography

[1] *Amazon Elastic Compute Cloud (Amazon EC2).* http://aws.amazon.com/ec2/.

[2] *Debian - The universal operating system.* http://www.debian.org.

[3] *FAI - Fully Automatic Installation.* http://fai-project.org/.

[4] *KVM - Kernel Based Virtual Machine.* http://www.linux-kvm.org.

[5] *Oracle Solaris 11 Express Automated Installer Guide Oracle Solaris 11 Express Automated Installer Guide Oracle Solaris 11 Express Automated Installer Guide Oracle Solaris 11 Express Automated Installer Guide.* http://download.oracle.com/docs/cd/E19963-01/html/820-6566/.

[6] *Welcome to xen.org, home of the Xen hypervisor, the powerful industry standrad for virtualisation.* http://xen.org/.

[7] *Windows Azure Platform.* http://www.microsoft.com/windowsazure/.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communicationmunication*, pages 63–74, Seattle, WA, USA, 2008.

[9] M. Al-Ghamdi, A. Chester, L. He, S. Jarvis, and J. Xue. Predictive and dynamic resource allocation for enterprise systems. In *Proceedings of the 10th International Conference on Scalable Computing and Communications*, 2010.

[10] M. Al-Ghamdi, A. Chester, and S. Jarvis. Predictive and dynamic resource application for enterprise applications. In *Proceedings of the 10th IEEE International Conference on Scalable Computing and Communications*, pages 2776–2783, June 2010.

[11] Y. An, T. Kin, T. Lau, and P. Shum. A scalability study for websphere application server and db2 universal database. Technical report, IBM, 2002.

[12] The Apache Software Foundation, http://cwiki.apache.org/GMOxDOC12/daytrader.html. *Daytrader*.

[13] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rockwerger. Oceano- SLA based management of a computing utility. In *Proceedings of the International Symposium on Integrated Network Management*, pages 855–868, 2001.

[14] D. Ardagna, M. Trubian, and L. Zhang. Sla based resource allocation policies in autonomic environments. *Journal of Parallel and Distributed Computing*, 67:259–270, 2007.

[15] M. Arlitt and T. Jin. A workload characterization of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, May 1999.

[16] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, University of California at Berkeley, 2009.

[17] D. Armstrong and K. Djemame. Performance issues in clouds: An evaluation of virtual machine propagation and i/o paravirtualisation. *The Computer Journal*, 54(6), June 2011.

[18] D. Bacigalupo, S. Jarvis, L. He, D. Spooner, and G. Nudd. Comparing layered queuing and historical performacne models of a distributed enterprise application. *Journal of Supercomputing*, 34(2):93–111, 2005.

[19] D. A. Bacigalupo, J. W. J. Xue, S. D. Hammond, S. A. Jarvis, D. N. Dillenberger, and G. R. Nudd. Predicting the effect on performance of container-managed persistence in a distributed enterprise application. In *IPDPS 07*. IEEE Computer Society Press, March 2007.

[20] J. Bacon, D. Evans, D. M. Eyres, M. Migliavacca, P. Pietzuch, and B. Shand. Enforcing end-to-end application security in the cloud. In *Proceedings of the 11th International Middleware Conference*. Springer, 2010.

[21] S. A. Banawan and N. M. Zeidat. A Comparitive Study of Load Sharing in Heterogeneous Multicomputer Systems. In *Proc. Annual Simulation Symposium*, pages 22–31, 1992.

[22] K. Bouyoucef, I. Limam-Bedhaif, and O. Cherkaoui. Optimal allocation approach of virtual servers in cloud computing. In *Proceedings of the 6th Euro-NF Conference on Next Generation Internet*, 2010.

[23] M. Burgess. A site configuraton engine. In *Computing Systems*, volume 8. The USENIX Association, Summer 1995.

[24] A. Celesti, F. Tusa, M. Villari, and A. Puliafito. How to enhance cloud architectures to enable cross-federationq. In *the 3rd IEEE Internationsal Conference on Cloud Computing*, pages 337–345, 2010.

[25] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the ACM*

SIGMETRICS International Conference on the Measurement and Modeling of Computer Systems, pages 300–301, June 2003.

[26] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proceedings of the 12th International Symposium on High Performance Distributed Computing*, pages 90–100, June 2003.

[27] L. Cherkasova and P. Phaal. Session Based Admission Control: a Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers*, 51(6), jan 2002.

[28] A. Chester, M. Leeke, M. Al-Ghamdi, A. Jhumka, and S. Jarvis. A framework for data centre scale dynamic resource allocation algorithms. In *Proceedings of the 11th International Conference on Scalable Computing and Communications*, 2011.

[29] A. Chester, J. Xue, L. He, and S. Jarvis. A system for dynamic server allocation in application server clusters. In *Proc. International Symposium on Parallel and Distributed Processing with Applications*, pages 130–139, December 2008.

[30] M. Crovella and A. Bestavros. Self-similarity in the world wide web: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1998.

[31] G. Cuomo. A methodology for performance tuning. Technical report, IBM, 2000.

[32] P. P. D. Villela and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Transaction on Internet Technology*, 7(1):7, 2007.

[33] J. Dean. Underneath the covers at google: Current systems and future directions. Presentation at Google IO Conference 2008, 2008.

[34] K. Dutta and A. D. et al. ReDAL: An Efficient and Practical Request Distribution Technique for Application Server Clusters. *IEEE transactions on Parallel and Distributed Systems*, 18(11):1516–1527, 2007.

[35] W. W. Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. In *Open Information Systems 10*, Jan. 1995.

[36] Eucalyptus Systems, Inc., http://www.eucalyptus.com/. *Eucalyptus: Leader in Enterprise Cloud Computing and Private Cloud Open Source Software.*

[37] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison Wesley, 2002.

[38] J. Franklin, A. Seshadri, N. Qu, S. Chaki, and A. Datta. Attacking, repairing and verifying secvisor: A reterospective on the security of a hypervisor. Technical Report CMU-CyLab-08-008, Carnegie Mellon University, June 2008.

[39] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centres: Measurements, analysis and implications. In *Proceedings of SIGCOMM 2011*, 2011.

[40] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the 10th International Symposium on Workload Characterization*, pages 171–180. IEEE, 2007.

[41] L. He, W. J. Xue, and S. A. Jarvis. Partition-based profit optimisation for multi-class requests in clusters of servers. In *Proc. International Conference on e-Business Engineering*, pages 131–138, October 2007.

[42] L. He, W. J. Xue, and S. A. Jarvis. Partition-based Profit Optimisation for Multi-class Requests in Clusters of Servers. In *the IEEE International Conference on e-Business Engineering*, 2007.

[43] J. Idziorek. Discrete event simulation model for analysis of horizontal scaling in the cloud computing model. In *Proceedings of the Winter Simulation Conference*, pages 3004–3014, December 2010.

[44] P. Joshi, H. S. Gunawi, and K. Sen. Prefail: Programmable and efficient failure testing framework. Ucb/eecs-2011-3, University of California at Berkeley, 2011.

[45] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *2010 International Conference on Distributed Computing Systems*, pages 62–73, 2010.

[46] A. Kochut. On impact of dynamic virtual machine reallocation on data center efficiency. In *Proceedings of 16th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecomunication Systems (MASCOTS)*, 2008.

[47] G. Lee, B.-G. Chun, and R. Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *3rd USENIX Workshop on Hot Topics in Cloud Computing*, 2011.

[48] S. Lim, B. Sharma, G. Nam, E.K.Kim, and C. Das. Mdcsim: A multi-tier data center simulation platform. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing*. IEEE, 2009.

[49] X. Liu, J. Heo, and L.Sha. Modeling 3-tiered web applications. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 307 – 310, May 2005.

[50] Z. Liu, M. Squillante, and J. Wolf. On Maximizing Service-level-agreement Profits. *ACM SIGMETRICS Performance Evaluation*, 29:43–44, 01.

[51] D. Menasce and V. Almeida. *Capacity Planning for Web Services*. Prentice Hall, 2002.

[52] D. Menasce, V. Almeida, and L. Dowdy. *Capacity Planning and Performance Modelling: From Mainframes to Client-Server Systems*. Prentice Hall, 1994.

[53] D. Menasce, V. Almeida, and et al. Business-oriented Resource Management Policies for e-Commerce Servers. *Performance Evaluation*, 42:223–239, 2000.

[54] OpenNebula Project Leads (OpenNebula.org), http://opennebula.org/. *OpenNebula: The Open Source Toolkit for Cloud Computing*.

[55] J. Palmer and I. Mitrani. Optimal and heuristic policies for dynamic server allocation. *Journal of Parallel and Distributed Computing*, 65(10):1204–1211, 2005.

[56] D. A. Patterson. A simple way to estimate the cost of downtime. In *Proceedings of the 16th USENIX Systems Administration Conference*, pages 185–188, November 2002.

[57] M. Reiser and S. Lavenberg. Mean-value analysis of closed multi-chain queuing networks. *Journal of the Association for Computing Machinery*, 27:313–322, 1980.

[58] D. Rice. An analytical model for computer system performance evaluation. *SIGMETRICS Performance Evaluation Preview*, 2:14–30, 1973.

[59] B. P. Rimal, E. Choi, and I. Lumb. A taxonomy and survey of cloud computing systems. In *the Fifth International Joint Conference on Networked Computing, Advanced Information Management and Digital Content and Multimedia Technologies*, 2009.

[60] R. Sailer, E. Valdez, T. jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. Technical report, IBM, 2005.

[61] P. Sempolinsky and D. Thain. A comparison and critique of eucalyptus, opennebula and nimbus. In *Proceedings of the 2nd International Conference on Cloud Computing Technology and Service*, 2010.

[62] N. G. Shivaratri, P. Krueger, and M. Shingal. Load Distribution for Locally Distributed Systems. *IEEE Computer*, 8(12):33–44, December 1992.

[63] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–10, May 2010.

[64] J. Slegers, I. Mitriani, and N. Thomas. Evaluating the optimal server allocation policy for clusters with on/off sources. In *Fourth European Performance Engineering Workshop, EPEW 2007, Berlin, Germany*, 2007.

[65] J. Slegers, I. Mitriani, and N. Thomas. Evaluating the optimal server allocation policy for clusters with on/off sources. *Journal of Performance Evaluation*, 66(8):453–467, August 2009.

[66] Sun Microsystems, Inc. *Sun Java System Application Server 9.1 Performance Tuning Guide*, 2007.

[67] G. Teodoro, T. Tavares, B. Coutinho, and W. M. amd D. Guedes. Load balancing on stateful clustered web servers. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, pages 207–215, 2003.

[68] G. Tian and D. Meng. Failure rules based node resource provisioning policy for cloud computing. In *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications*, pages 397–404, December 2010.

[69] K. Ueno and T. A. et al. Websphere scalability: Wlm and clustering, using websphere application server advanced edition (ibm redbook). Technical report, IBM, September 2000.

[70] University of Chicago, http://www.nimbusproject.org/. *Nimbus*.

[71] B. Urgaonkar, G. Pacifi, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems*, June 2005.

[72] B. Urgaonkar, P. Shenoy, A. Chandra, and et al. Dynamic Provisioning of Multi-tier Internet Applications. In *Second International Conference on Autonomic Computing (ICAC'05)*, 2005.

[73] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, January 2008.

[74] A. Verma, P. De, V. Mann, T. Nayak, A. Purohit, G. Dasgupta, and R. Kothari. Brownmap: Enforcing power budget in shared data centers. In *Proceedings of the 11th International Middleware Conference*, 2010.

[75] A. Verma, G. Kumar, and R. Koller. The cost of reconfiguration in a cloud. In *11th International Middleware Conference*. ACM, 2010.

[76] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 193–204, June 2010.

[77] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Proceedings of the 1st International Conference on Cloud Computing and Science (CloudCom 2009)*, 2009.

[78] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In *the 2003 USENIX Symposium on Internet Technologies and Systems*, 2003.

[79] M. Wiboonrat. An empirical study on data center system failure diagnosis. In *Proceedings of the 3rd International Conference on Internet Monitoring and Protection*, pages 103–108, June 2008.

[80] J. Xue, A. Chester, L. He, and S. Jarvis. Dynamic resource allocation in enterprise systems. In *Proceedings of the 14th International Conference on Parallel and Distributed Systems*, pages 203–212, December 2008.

[81] W. Zhao and H. Schulzrinne. Predicting the upper bound of web traffic volume using a multiple time scale approach. WWW2003 Poster.